ECE 385

Fall 2022

Final Project

Implementing a Mining Simulator Video Game

Ravi Thakkar, Siraj Khogeer

TA: Gavin Wu

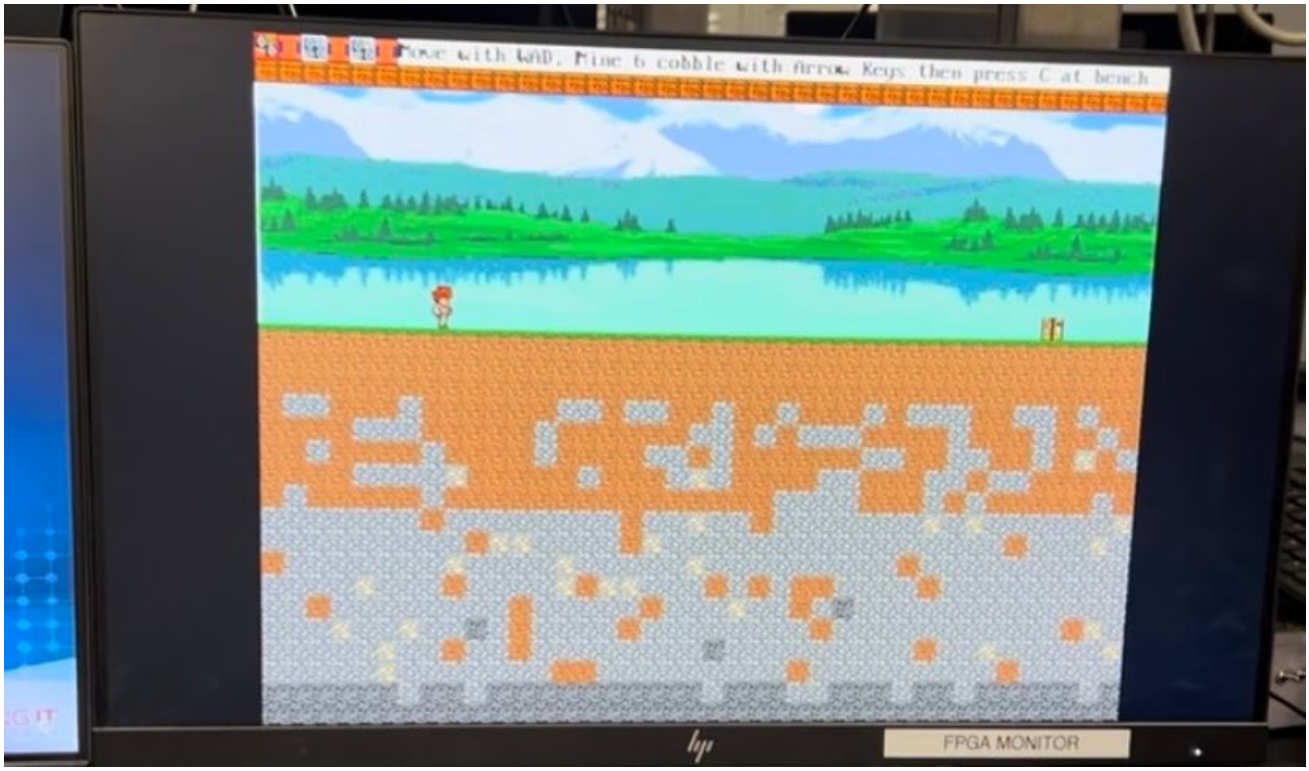
12-10-2022

Introduction

a. Summarize functionality of video game

The video game we made is an amalgam between Minecraft and Terraria. There are two main functionalities for this game: the ability for our player to build and mine blocks in the world in a Minecraft like fashion, and have a sort of progression through upgrades next to the crafting table. From this starting point we were able to implement many things. The game starts by providing instructions to the player through the font_rom, and guides him to mine blocks to be able to build better pickaxes to mine more difficult blocks to finally win the game.

The blocks essentially are the world itself apart from the background, and we pseudo-randomly generated the world using four types of blocks: dirt, stone, gold ore, and bedrock. We also wanted to create some sort of layers to mimic the sort of stratification of blocks that you would see in Minecraft.



Pseudorandomly Generated World During Final Lab Demo. Instructions On Top Of Screen.

b. Summarize how major features were implemented, including vga, keyboard, sprites, audio

We implemented this module by using the AVL_text_interface as the starting point. The first major feature to be implemented is the world. The world was implemented by separating the screen into 16x16 pixel blocks, and the index of each block was stored on OCM. If the index was

0, then it was assumed that there was no block present. We then decided to implement the background image.

We then implemented the background image by using RAM that held a hex file. From here we then went to implement the sprites for the blocks. We used an OCM ram to hold the sprite data for each sprite index. From this we were able to draw the blocks of the index!=0, using the sprite ram, and then the background if the index===0;

We then implemented the character movement utilizing a modified version of the ball module, and in conjunction with that, we made a draw module which took care of drawing the character and animating it depending on the movement. The character could move, jump, and mine. We implemented animations for the movement and left and right character sprites by using a sprite RAM that was written to using NIOS. From these three sprites we were able to draw a screen by having a hierarchy char>block>background.

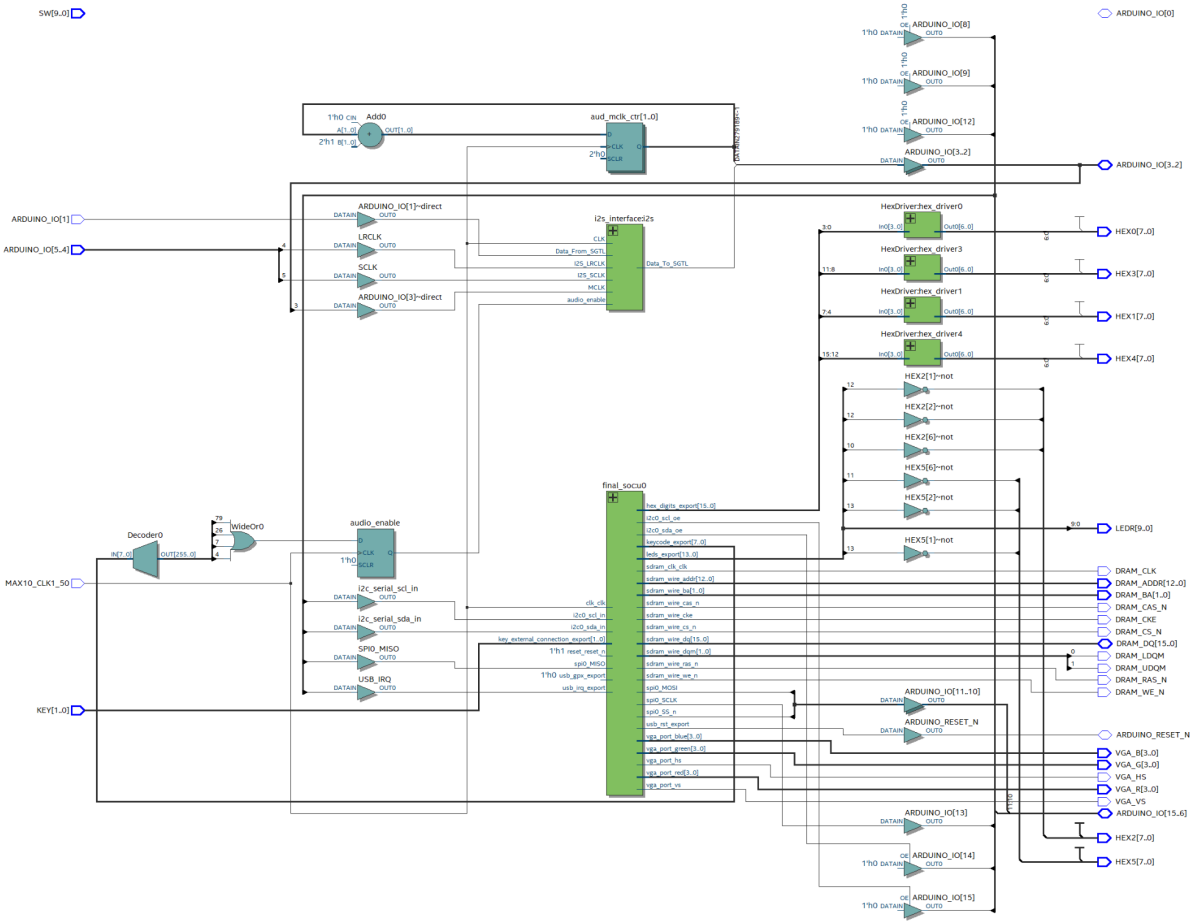
We implemented the vga by using the VGA controller and exporting it to the top level module. We then used NIOS to get the keycodes from the keyboard through the SPI protocol and wrote that to register in the avl_text_intertface peripheral which then used it to control the animations and the movement. We then took care of the world logic such as the mining, building, and crafting through case statements and if statements in C.

The audio was implemented by adding an additional peripheral to the SoC, which was an I2C Master IP for the FPGA. This IP Block allowed for control over the SGTL5000 Codec and allowed for data to be passed into the Codec using the arduino pins. An I2S interface used clocks generated by the SGTL and established a bit stream to I2S_DIN from a file containing data from an audio. The next thing that needed to be done was taking a small wav file and converting the data to 16-bit signed hex, then taking that data and creating a ROM module that could contain the data in order to be accessed by the I2S interface. We used CA Zayd's helper function to generate a txt file from a wav input as well as the ROM module. Since our audio was a walking sound effect from Minecraft, we needed to only allow sound to be played when one of the 'WASD' keys were pressed. This was implemented by having an audio_enable signal that would turn high and allow the bit-stream to pass into I2S_DIN when one of the 'WASD' keys were pressed.

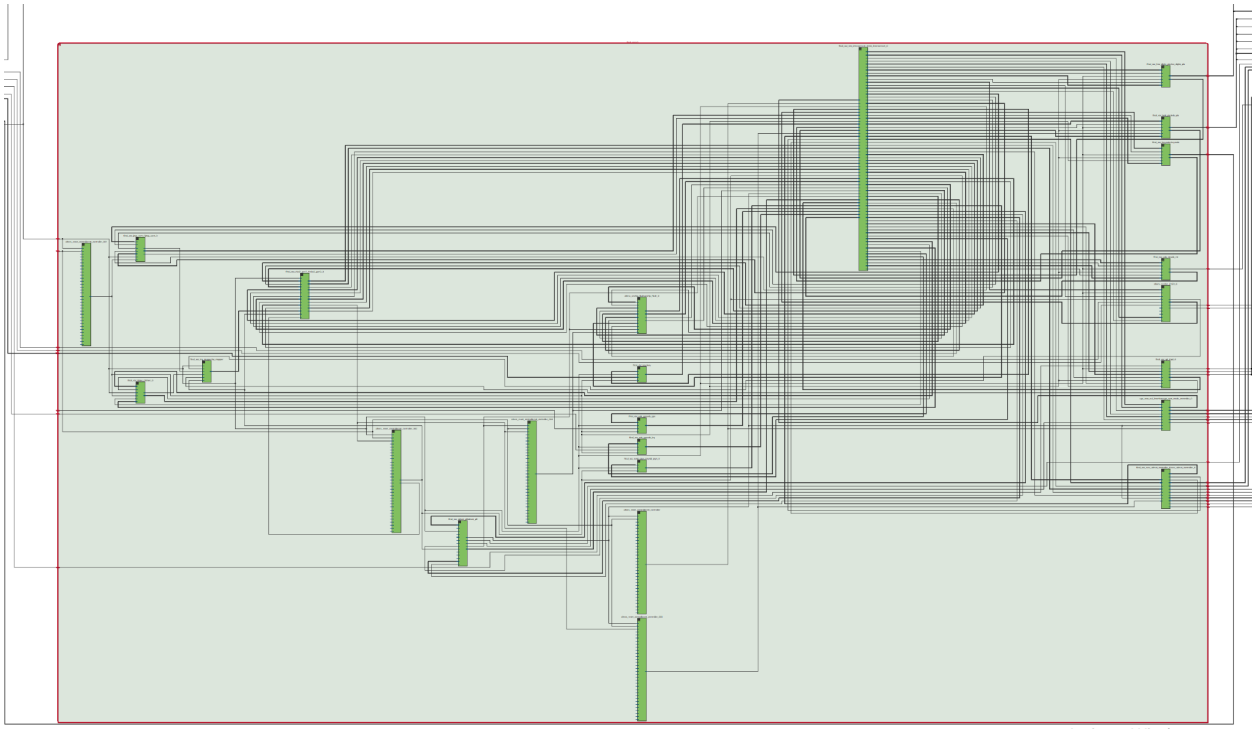
Written Description of Final Project

Block Diagram

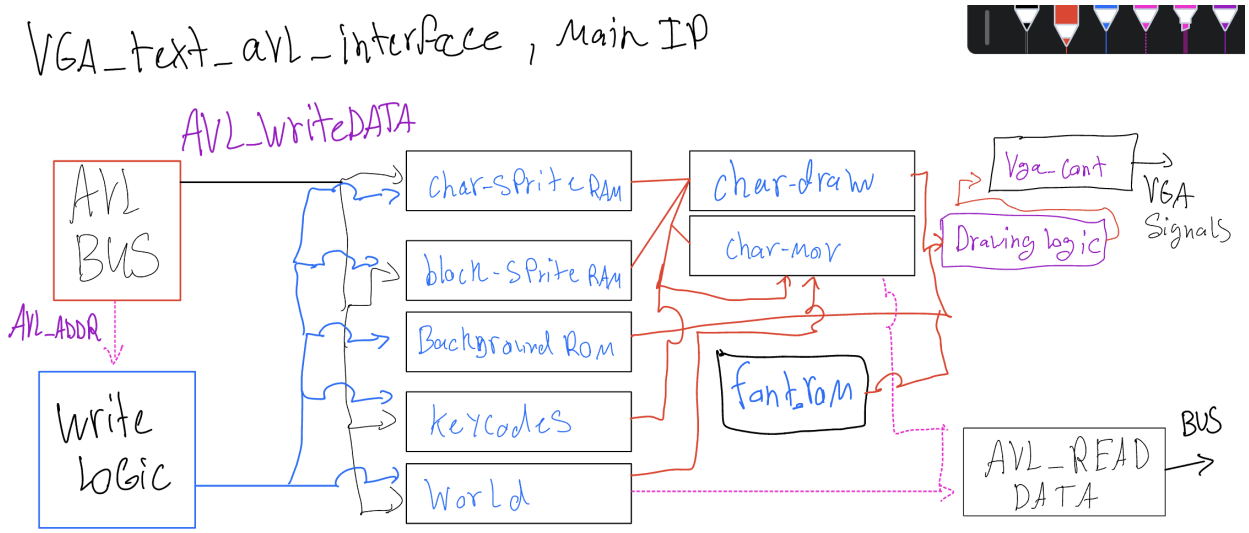
a. This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module.



Top Level Block Diagram



SOC Block Diagram



Simplified Block Diagram: *All Memories have different addresses depending on char (x,y) or DrawX and DrawY*

b. Note that depending on your layout of the registers inside your main module, the Quartus view may be illegible, in which case you should draw a block diagram using software.

You may start from the provided materials (e.g. in IAMM), but you should fill in the specific signals between the modules and the inside subcomponents within each module.

c. You should have block diagrams for both the Week 1 and Week 2 portions, a good setup is to show the common components (e.g. the SoC setup) first and then show diagrams for both the Week 1 and Week 2 VGA controller components.
d. If your design has a state machine, you should include a State Diagram as well.

Written Description of .v and .sv Modules

Note: All Memories is run at 200mhz using CLK_v from the PLL

Module: **finalproject** in finalproject.sv

In/Outs:

```
//////// Clocks ////////
input     MAX10_CLK1_50,

//////// KEY ////////
input     [ 1: 0]   KEY,

//////// SW ////////
input     [ 9: 0]   SW,

//////// LEDR ////////
output    [ 9: 0]   LEDR,

//////// HEX ////////
output    [ 7: 0]   HEX0,
output    [ 7: 0]   HEX1,
output    [ 7: 0]   HEX2,
output    [ 7: 0]   HEX3,
output    [ 7: 0]   HEX4,
output    [ 7: 0]   HEX5,

//////// SDRAM ////////
output              DRAM_CLK,
output              DRAM_CKE,
output    [12: 0]   DRAM_ADDR,
output    [ 1: 0]   DRAM_BA,
inout     [15: 0]   DRAM_DQ,
output              DRAM_LDQM,
output              DRAM_UDQM,
output              DRAM_CS_N,
output              DRAM_WE_N,
output              DRAM_CAS_N,
output              DRAM_RAS_N,

//////// VGA ////////
output              VGA_HS,
output              VGA_VS,
output    [ 3: 0]   VGA_R,
output    [ 3: 0]   VGA_G,
output    [ 3: 0]   VGA_B,

//////// ARDUINO ////////
inout     [15: 0]   ARDUINO_IO,
inout               ARDUINO_RESET_N
```

Description: Top level which instantiates the SOC for the main program. It sets the control signals such as the Aurdiono and USB signals for the usb keyboard and the I2C. It also connects the inputs and outputs of the SOC such as the buttons and VGA outputs. This module also set audio enable to high when the character is moving, generates the clock for the SGTL5000, and sets the arduino pins to allow for audio clocks and data to be sent and received by the SGTL5000 Codec.

Purpose: Top level module, used for pin assignments, arduino/audio and usb pin assignments. Use the top level to instantiate SOC and set all control signals. Also used for audio enabling when moving.

Module: **vga_text_avl_interface** in vga_text_avl_interface.sv

Inputs: CLK, CLK_v, RESET, AVL_READ, AVL_WRITE, AVL_CS, AVL_BYTE_EN[1:0], AVL_ADDR[16:0], AVL_WRITEDATA[15:0]

Outputs: AVL_READDATA[15:0], red[3:0], green[3:0], blue[3:0], hs, vs

Description: This is the main module that is used in the project. This is an IP peripheral connected to the SOC through AVALON, and is controlled through NIOS. There exists wren logic by looking at the three MSB and deciding which OCM to write to. From this the sprites for the blocks and characters are loaded. The palette for the background is also loaded into registers using AVL. The AVL is then used to read and write the current world by looking using address of (col+row*40) to find what the index of the block is there. This is the base case. Another case is when the MSB is 1, the AVL_READATA reads the current x and y coordinates of the character. Finally the AVL is also used to write the keycodes from SPI into the register when only AVL[15]==1. The logic is clearly seen in the actual code, where wren vars are assigned to each different memory. This module also instantiates the VGA_controller which provides the pixel clock and HS/VS. From here the module instantiates char_mov and char_draw which are used for the character movement and drawing. The module also has a memory which holds the text to be displayed on the screen, this then feeds to a font_rom which is then connected to the output logic. Finally, a if statement is used to decide whether the instruction text, character sprite, block, sprite, or background should be drawn and they are exported to the RGB of the VGA. See below:

```
always_ff @(posedge CLK_v) begin

   if(DrawX>=96 && DrawY<16) begin
      if(data[8-DrawX%8]==1)
         color = 0;
      else
         color = 16'hffff;

   end else begin
   if(colorchar!=0)
      color = colorchar;
   else if(indexs!=0)
      color = colorsprite;
   else
      color = colorg;
   end
end
```

Purpose: This is used as the main module which handles the world, character, and drawing. It instantiates multiple OCMs, and other SV modules.

Module: **charmov** in charmov.sv

```
input logic [16:0] AVL_ADDR,
input logic [15:0] AVL_WRITEDATA,
input CLK_v, Reset, frame_clk, wrenfb,
input [15:0] keycode,
output [9:0]  BallX, BallY, BallS,
output logic[4:0]jumpf);
```

Description: This module controls the movement of the ball. The module first copies the world into 10 different 1-bit wide OCMs which hold the blocks in the current world. The character is

then created at a starting coordinates and is defined to be 15x31 pixels. From this the OCMs are read for the blocks above the character, below, and each side using a different variations of this formula: ((Ball_X_Pos - 8)/16)+((Ball_Y_Pos-15)/16)*40. If the data!=0 then the module knows that there is a block in that location. From this a default setting of BallY = BallY+1 is used to implement gravity. Two case statements are then used for the movement of the character. First the E key is used to start the game which causes the case statements to work. If A or D is pressed then the character moves in that direction unless there is a block or the edge of the screen in that direction. If W is pressed, the character jumps, which means that for 2^5 clock cycles the character moves up unless there is the top of the screen or a block there. The collision is done by reading the OCM and assigning flags for each possible side of the character, 10 in total. After the jump counter finishes the gravity turns back on. The character only changes location at the frame_clk which corresponds to VS. If the character is currently jumping then the counter cannot be interrupted. The jump counter is then exported to be used in animation, and the character X and Y coordinates are also outputted.

Purpose: This module is an improvement from the ball module, it controls the character movement and collision detection. This includes the implementation of gravity, jumping, and non-bounce collision detection

Module: **chardraw** in chardraw.sv

```
input        [9:0] BallX, BallY, DrawX, DrawY, Ball_size,
     input logic wren,
     input logic CLK,
     input logic left, hit, mov, jump,
     input logic[1:0] count,
     input  logic [1:0] AVL_BYTE_EN,          // Avalon-MM Byte Enable
     input  logic [16:0] AVL_ADDR,          // Avalon-MM Address
     input  logic [15:0] AVL_WRITEDATA,
    output logic [15:0]  colorchar );
```

Description: This module first takes the AVL signals, and if wren is high, it writes from the NIOS system onto the charram2 which is a 16-bit 16*16*32 OCM. The module then looks at the current x and y coordinates and compares them with the character coordinates, and if there is an overlap, the module outputs the color stored in the memory. To read from the memory the formula : (DrawX-BallX+7)+(DrawY-BallY+15)*16 + mov*512 + count*512 + jump*2560+hit*3072 + left*3584) is used. Essentially if the input left is high then the address increases to 3584 which is where the first left looking sprite is located, if mov is high then 512 is added which changes the sprite to that of a running character. To implement animations, if mov is high a 2-bit country is run and multiplied by 512 to give the next frame in the animation. If the jump signal is high then the address increments by 2560 and if there is a hit then by 3072. There is a hierarchy where hit goes before jump before mov before static.

Purpose: This module is used to draw the character sprite and animations.

Module: **i2s_interface** in 12s_interface.sv

```
input CLK,
input audio_enable,
input Data_From_SGTL,
input MCLK,
input I2S_LRCLK,
input I2S_SCLK,
output Data_To_SGTL
```

Description: The module takes in multiple clock inputs, the audio_enable signal, Data_From_SGTL (which is unused for our implementation), and outputs Data_To_SGTL which is one bit for every I2S_SCLK pulse. The ROM for any of our audio data is instantiated, and the module takes 32 bits of information at a time from the ROM using the negative edge of the LRCLK, then sends out two copies of the 32 bits (one copy for the left side, then one copy for the right), so there are 64 SCLK pulses for every negative edge of LRCLK which allow for Data_To_SGTL to be updated with a new bit. The audio_enable is multiplied by the bit sample that is being outputted to Data_To_SGTL, so when audio_enable is low (0), Data_To_SGTL is just a stream of zeroes, so audio is turned off. When audio_enable is high(1), Data_To_SGTL gets the sample bit, so in a way it acts like an AND gate to make sure that audio is only on when a key is being pressed.

Purpose: This module is necessary because it is synchronizing and formatting the audio data stored in a ROM correctly so it can be read by the SGTL, and then sending out the data bit-by bit.

Memory Modules:

```
module worldRAM3 (
    address_a,
    address_b,
    byteena_a,
    clock,
    data_a,
    data_b,
    wren_a,
    wren_b,
    q_a,
    q_b);

    input [10:0]  address_a;
    input [10:0]  address_b;
    input [1:0]  byteena_a;
    input   clock;
    input [15:0]  data_a;
    input [15:0]  data_b;
    input   wren_a;
    input   wren_b;
    output  [15:0]  q_a;
    output  [15:0]  q_b;
```

OCM used to hold the current blocks in the world, used in conjunction with the AVL_BUS to read and write the world

```
module sprite_ram2 (
    byteena_a,
    clock,
    data,
    rdaddress,
    wraddress,
    wren,
    q);

    input [1:0]  byteena_a;
    input   clock;
    input [15:0]  data;
    input [12:0]  rdaddress;
    input [12:0]  wraddress;
    input   wren;
    output  [15:0]  q;
```

used to hold block sprite data

```
module charram2 (
    byteena_a,
    clock,
    data,
    rdaddress,
    rden,
    wraddress,
    wren,
    q);

    input [1:0]  byteena_a;
    input   clock;
    input [15:0]  data;
    input [13:0]  rdaddress;
    input   rden;
    input [13:0]  wraddress;
    input   wren;
    output  [15:0]  q;
```

used to hold the character sprite data

```
module textRAM (
    address_a,
    address_b,
    clock,
    data_a,
    data_b,
    wren_a,
    wren_b,
    q_a,
    q_b);

    input [6:0]  address_a;
    input [6:0]  address_b;
    input   clock;
    input [7:0]  data_a;
    input [7:0]  data_b;
    input   wren_a;
    input   wren_b;
    output  [7:0]  q_a;
    output  [7:0]  q_b;
```

used to hold the text written by the NIOS be drawn

```
module backgroundRAM (
    address,
    clock,
    data,
    wren,
    q);

    input [16:0]  address;
    input   clock;
    input [7:0]  data;
    input   wren;
    output  [7:0]  q;
```

Background Rom used to draw background, initialized with HEX file.

```
module worldRAM2 (
    address_a,
    address_b,
    clock,
    data_a,
    data_b,
    wren_a,
    wren_b,
    q_a,
    q_b);

    input [10:0]  address_a;
    input [10:0]  address_b;
    input   clock;
    input [0:0]  data_a;
    input [0:0]  data_b;
    input   wren_a;
    input   wren_b;
    output   [0:0]  q_a;
    output   [0:0]  q_b;
```

1-bit wide OCM used to hold world data, used for collision detection

Hex Files:

Backgroundfinal.hex : used to initialize the background RAM to draw the screen and save on memory, stored in 320x240 palleted format.

Note:

The character sprites, palette, and block sprites are stored in a C header file.

Module: **HexDriver** in HexDriver.sv

Inputs: In0[3:0]

Outputs: Out0[6:0]

Description: HexDriver is used to map the inputs to outputs that can be seen on the hex display.

Purpose: Maps the input to outputs that can be displayed on the HEX display.

Module: **font_rom** in font_rom.sv

Inputs: addr[10:0]

Outputs: data[7:0]

Description: This module is a ROM that holds the data for all the 128 different characters/glyphs to be drawn. Each glyph has 16 lines that have 8 bits of data each which correspond to whether the pixel should be lit up or not. The address is found by doing 16*index+Y position. The resultant data is then drawn from left to right which needs a 8-DrawX%8 due to the endian discrepancy.

Purpose: This module was used to draw text on the screen in a legible format from NIOS to give instructions to the player.

Module: **walkcut_2_mono_rom** in walkcut_2_mono.sv

```
input clk,
input [13:0] addr,
output logic [31:0] q
```

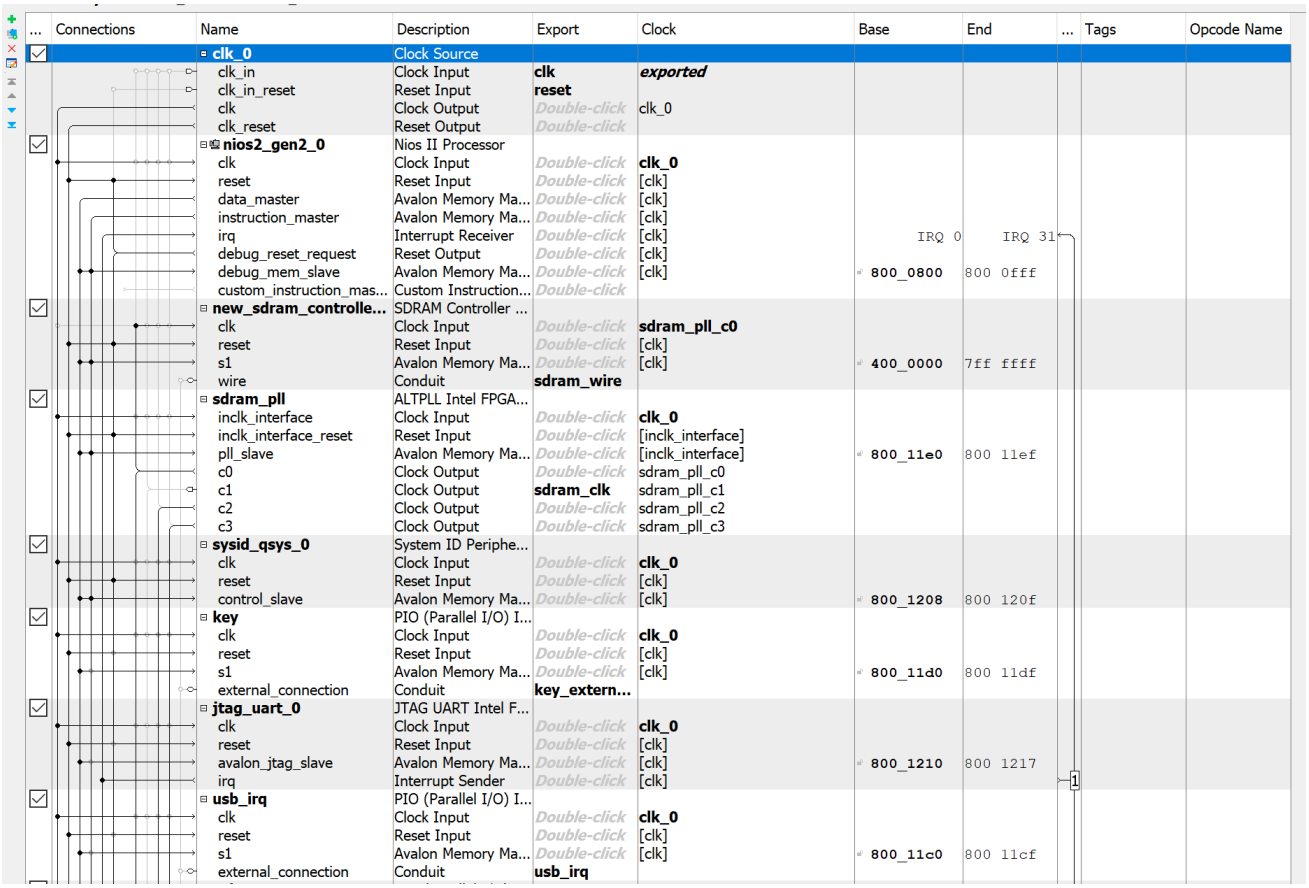
Description: This is the ROM SV file that was generated through CA Zayd's helper program in Python. The ROM itself contains all the data for the sound effect audio and stores it in exactly 13.167 registers that each contain 24 bits of data, which are 24-bit signed numbers that are used to generate the audio by the codec. The large amount of registers is surprisingly just for about ~0.3 seconds of audio, but it makes more sense when you take into consideration that we were using audio sampled at 44.1kHz.

The module reads a text file of the same name (also generated by the helper program, using a .wav file as an input) with all of the data in hex and writes to all of the registers. The input addr can address each of these registers in order to get the 24 bits of data stored in each of them. Once a specific register is address, the signal is concatenated with one byte as per the guidelines to have a dummy byte preceding the 24 bits of data that the SGTL reads. We made a couple modifications above the autogenerated file that was provided by Zayd's helper program: first was ensuring that the output would be 32 bits as required by concatenation, and second was fixing simple errors that were present in the SystemVerilog code.

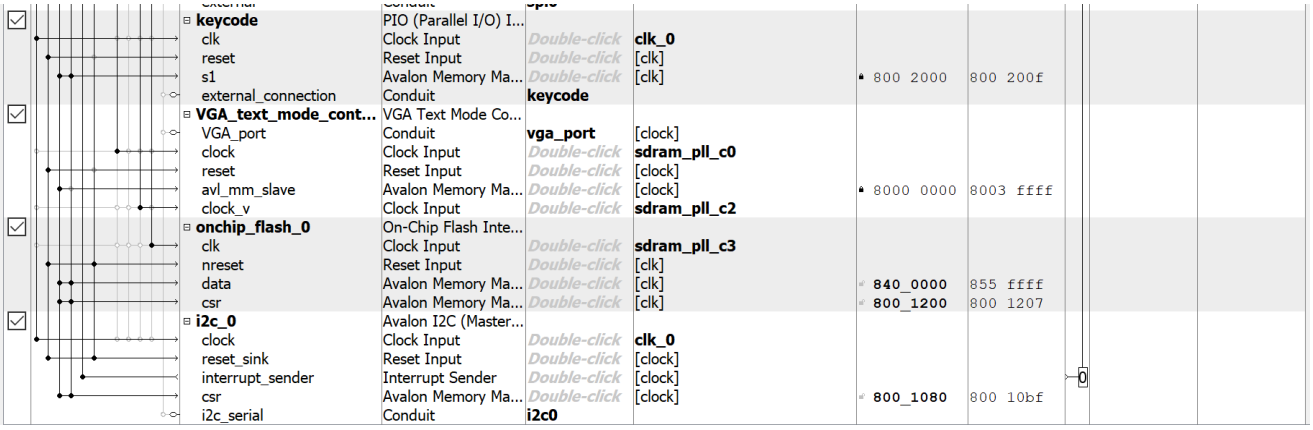
Purpose: The purpose of this module is to have a simple method of storing and retrieving the audio data through the I2S interface. Similar versions of this module can be generated for any audio file or waveform.

System Level Block Diagram

Final Project Platform Designer View:



...	Connections	Name	Description	Export	Clock	Base	End	...	Tags	Opcode Name
		clk_0	Clock Source							
		clk_in	Clock Input	**clk**	***exported***					
		clk_in_reset	Reset Input	**reset**						
		clk	Clock Output	*Double-click*	clk_0					
		clk_reset	Reset Output	*Double-click*						
		nios2_gen2_0	Nios II Processor							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		data_master	Avalon Memory Ma...	*Double-click*	[clk]					
		instruction_master	Avalon Memory Ma...	*Double-click*	[clk]					
		irq	Interrupt Receiver	*Double-click*	[clk]	IRQ 0	IRQ 31			
		debug_reset_request	Reset Output	*Double-click*	[clk]					
		debug_mem_slave	Avalon Memory Ma...	*Double-click*	[clk]	**800_0800**	800 0fff			
		custom_instruction_mas...	Custom Instruction...	*Double-click*						
		new_sdram_controlle...	SDRAM Controller ...							
		clk	Clock Input	*Double-click*	**sdram_pll_c0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	**400_0000**	7ff ffff			
		wire	Conduit	**sdram_wire**						
		sdram_pll	ALTPLL Intel FPGA...							
		inclk_interface	Clock Input	*Double-click*	**clk_0**					
		inclk_interface_reset	Reset Input	*Double-click*	[inclk_interface]					
		pll_slave	Avalon Memory Ma...	*Double-click*	[inclk_interface]	**800_11e0**	800 11ef			
		c0	Clock Output	*Double-click*	sdram_pll_c0					
		c1	Clock Output	**sdram_clk**	sdram_pll_c1					
		c2	Clock Output	*Double-click*	sdram_pll_c2					
		c3	Clock Output	*Double-click*	sdram_pll_c3					
		sysid_qsys_0	System ID Periphe...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		control_slave	Avalon Memory Ma...	*Double-click*	[clk]	**800_1208**	800 120f			
		key	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	**800_11d0**	800 11df			
		external_connection	Conduit	**key_extern...**						
		jtag_uart_0	JTAG UART Intel F...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		avalon_jtag_slave	Avalon Memory Ma...	*Double-click*	[clk]	**800_1210**	800 1217			
		irq	Interrupt Sender	*Double-click*	[clk]			1		
		usb_irq	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	**800_11c0**	800 11cf			
		external_connection	Conduit	**usb_irq**						



		keycode	PIO (Parallel I/O) I...							
		clk	Clock Input	*Double-click*	**clk_0**					
		reset	Reset Input	*Double-click*	[clk]					
		s1	Avalon Memory Ma...	*Double-click*	[clk]	800 2000	800 200f			
		external_connection	Conduit	**keycode**						
		VGA_text_mode_cont...	VGA Text Mode Co...							
		VGA_port	Conduit	**vga_port**	[clock]					
		clock	Clock Input	*Double-click*	**sdram_pll_c0**					
		reset	Reset Input	*Double-click*	[clock]					
		avl_mm_slave	Avalon Memory Ma...	*Double-click*	[clock]	8000 0000	8003 ffff			
		clock_v	Clock Input	*Double-click*	**sdram_pll_c2**					
		onchip_flash_0	On-Chip Flash Inte...							
		clk	Clock Input	*Double-click*	**sdram_pll_c3**					
		nreset	Reset Input	*Double-click*	[clk]					
		data	Avalon Memory Ma...	*Double-click*	[clk]	**840_0000**	855 ffff			
		csr	Avalon Memory Ma...	*Double-click*	[clk]	**800_1200**	800 1207			
		i2c_0	Avalon I2C (Master...							
		clock	Clock Input	*Double-click*	**clk_0**					
		reset_sink	Reset Input	*Double-click*	[clock]					
		interrupt_sender	Interrupt Sender	*Double-click*	[clock]			0		
		csr	Avalon Memory Ma...	*Double-click*	[clock]	**800_1080**	800 10bf			
		i2c_serial	Conduit	**i2c0**						

Functionality of IP Blocks

IP Block: **clk_0** Functionality: This is the IP block that contains a clock output for the rest of the modules. It is set to 50 MHz. Its input comes from the FPGA oscillator.
IP Block: **nios2_gen2_0** (NIOS II Processor) Functionality: This is the CPU that is used in this lab. It is in charge of the accumulation circuit

and the USB/SPI communication. Main CPU where the C code runs on. Also used to access different parts of the Avalon Bus through the C code.
IP Block: **sdram** Functionality: Memory used to store information for the CPU. Maps to the SDRAM on the FPGA. Used as high capacity memory for the CPU to store and run programs.
IP Block: **sdram_pll** Functionality: This module is used to create a clock signal that is out of phase from the main clock at a specific angle. This is used to drive the SDRAM controller and the FPGA SDRAM at different clocks. This skew is needed due to the precise timing of the SDRAM and is used to compensate for the delay and inaccuracies between the SDRAM and the controller. This was also used to drive the OCM at 200mhz in our implementation to reduce lag/glitches.
IP Block: **sysid_qsys_0** Functionality: This is used to ensure compatibility between the hardware and software components, for example the NIOS system and the loaded program on the FPGA. Gives a warning when there is a mismatch of the serial number provided. Used for warning and debugging purposes.
IP Block: **key** Functionality: This module is a PIO, a parallel IO module. Specifically this is a 2-bit input module that is mapped to the buttons on the FPGA. This is used for the accumulation circuit, where run is needed to be pressed to see when to accumulate, and reset in some other cases.
IP Block: **jtag_uart** Functionality: This module is used to communicate with the NIOS system while it's running through the computer terminal. This allows statements such as print and scan to work in C, and is used mainly for debugging.
IP Block: **keycode** Functionality: This is a PIO block. This is a 8-bit output port that is a part of the SOC. This is used to export the read character from the keyboard to the top-level sv module which then goes into the ball module.
IP Block: **usb_irq** Functionality: This is a 1-bit PIO block. It is in input mode. This is used as an interrupt port that the MAX chip uses. Used for interrupt functionality in some cases. This is mapped to the chip via the top sv module and the FPGA pin assignments.
IP Block: **usb_gpx** Functionality: This is a 1-bit input PIO block. It is required for the MAX chip, and is a general purpose push pull output. Functionality is decided by the chip and depends on different factors. This is mapped to the chip via the top sv module and the FPGA pin assignments.
IP Block: **usb_rst** Functionality: This is a 1-bit output PIO block. It is required for the MAX chip. This is

mapped to the chip via the top sv module and the FPGA pin assignments.
IP Block: **hex_digits_pio** Functionality: This is a 16-bit output PIO block. This is used to output the hex of the keycode. Used to map the FPGA ports to the SOC in the top-level to display on the HEX display. The NIOS system sets the output according to the current keyboard presses.
IP Block: **leds_pio** Functionality: This is a 14-bit output PIO block. This is used to display the LED accumulation on the FPGA leds. Mapped to the ports of the FPGA from the top-level SOC.
IP Block: **timer_0** Functionality: This is a timer that keeps track of the timeouts that the USB requires. This is required for the USB SPI implementation on the MAX chip. An interrupt is assigned which is connected to the NIOS system.
IP Block: **spi_0** Functionality: This is the SPI port peripheral that is required to use SPI between the NIOS systema and the MAX chip. In this case the NIOS system is the host and MAX is the slave. The SPI ports are connected via the top-level to the MAX chip via the FPGA.
IP Block: **VGA_text_mode_controller** Functionality: This is the main IP used in this project. The NIOS system interacts with this to write the world through the memory that is held in this IP block. This takes care of the movement, drawing, and collision detection of the game.
IP Block: **on_chip_flash** Functionality: Un-used
IP Block: **i2c_0** Functionality: The I2C Master IP block allows for control over the SGTL 5000 Codec. After being added to Platform Designer, we added the input signals for the I2C IP Block on the top level module, finalproject.sv. This meant that the SDA and SCL lines that are inbound and outbound from the master and the SGTL device needed to be connected with some of the arduino pins, which was a step that was mostly provided to us. Once that was complete, when generating the BSP on the NIOS, header files which provide necessary I2C commands are generated so that the SGTL can be interfaced using C code on the NIOS.

NIOS:

Note: Re-Used C files from previous labs are not included, note that VRAM stands for the AVL_VGA_interface IP, where VRAM[ADDR] = VRAM[AVL_ADDR]

File: background12.h

Functionality: This file has the palette that the program uses to draw the background image. It is stored in a char array format.

```
static unsigned char header_data_cmap[256][3] = {
    { 21,  8,  8},
```

File: sprites.h
Functionality: This files holds all the sprites data in the form of structs. This is then used to load the OCM on the SOC/IP peripheral, main SV module. Each address of data hold either R G or B where each is a byte/8-bits

```
static const struct {
  unsigned int   width;
  unsigned int   height;
  unsigned int   bytes_per_pixel; /* 2:RGB16, 3:RGB, 4:RGBA */
  unsigned char  pixel_data[16 * 31 * 3 + 1];
} charmov3v2 = {
  16, 31, 3,
  "\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\063\026\015\000\000\000L\040\023L\040\023L\040\023\000\000\000"
```

File: main.c
Functionality: This is the main program on the NIOS system, this takes care of the USB communication, audio, and game logic, and sprite and memory loading.

The main program is composed of three main parts, the audio initialization, the USB initialization, and then the game infinite while loop.

Other than the main function there are four modified functions. Note that the repeated or given functions are not included.

Audio_main: This is the code copied from the sgtl_test.c file that allows us to use I2C commands to interface with the SGTL. This function is invoked once in main() in order to configure the SGTL to match our implementation.

USB_in: This function is given with the USB code provided, the main function calls this first to initialize the SPI and USB protocol and devices.

USB_run: This is what used to be the while loop in the given USB code, it was modified so that after the keycode is set, before the function sets the hex display, the NIOS writes the keycodes into the VRAM then returns. The while loop runs normally and goes through all the USB tasks until it reaches the end where it is broken by using a return function to return to the game loop.

```
printf("keycodes: ");
for (int i = 0; i < 6; i++) {
    printf("%x ", kbdbuf.keycode[i]);
}
setKeycode(kbdbuf.keycode[0]);

vga_ctrl->VRAM[0x08000] = kbdbuf.keycode[0];
vga_ctrl->VRAM[0x08001] = kbdbuf.keycode[1];
return kbdbuf.keycode[1]<<8 | kbdbuf.keycode[0];
printSignedHex0(kbdbuf.keycode[0]);
printSignedHex1(kbdbuf.keycode[1]);
printf("\n");
```

void mine(int* dirti, int* cobblei, int* goldi, int block, int prog):

This function was created to reduce clutter in the code. This takes care of the mining mechanic. The inputs dirti, cobblei, and goldi are pointers to the variables holding the inventory value of each block. block is used as the address of the current arrow key being pressed. It corresponds to a block in the worldRAM. prog is used to track the progression of the game, ie what pickaxe the player has, so which blocks he can mine.

Then the function is composed of three if statements with a case statement in each. The function first checks if there is a block in the given address “block”. If there is a block then two if statements are used to see the progression of the game. If prog =0, then the character can mine dirt in 1 hit, cobble in 2, and not gold. If prog = 1 then the character can mine dirt in 1 hit, cobble in 1, and gold in 2, if prog = 2 then dirt, gold, and cobble can be mine in 1 hit. No other block can be mine. Note that when a block is half broken the index of the block is increased by 3 to show a half broken block. If a block is mined then vga_ctrl->VRAM[block] = 0; is used, or a half block is set correspondingly.

```
if(prog==0){
    switch(vga_ctrl->VRAM[block]){
                        case    1 :
                            vga_ctrl->VRAM[block] = 0;

                            *dirti = *(dirti)+1;

                            for(int row=0; row<8; row++){
                                for(int col=0; col<7; col++){
                                    r = numbers8.pixel_data[3*(col+*(dirti)/10*7)+210*row];
                                    g = numbers8.pixel_data[3*(col+*(dirti)/10*7)+210*row+1];
                                    b = numbers8.pixel_data[3*(col+*(dirti)/10*7)+210*row+2];
                                    //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
                                    vga_ctrl->VRAM[0x08000+4096+2*(col+1)+(row+1)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);
                                    vga_ctrl->VRAM[0x08000+4096+2*(col+1)+(row+1)*32+1] = (r/16)&0x0F;

                                    r = numbers8.pixel_data[3*(col+(*(dirti)-*(dirti)/10*10)*7)+210*row];
                                    g = numbers8.pixel_data[3*(col+(*(dirti)-*(dirti)/10*10)*7)+210*row+1];
                                    b = numbers8.pixel_data[3*(col+(*(dirti)-*(dirti)/10*10)*7)+210*row+2];

                                    vga_ctrl->VRAM[0x08000+4096+2*(col+8)+(row+7)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);
                                    vga_ctrl->VRAM[0x08000+4096+2*(col+8)+(row+7)*32+1] = (r/16)&0x0F;

                                }
                            }
```

After the block is mined, the inventory is updated by using the given formula/function. This essentially draws a number between 0-9 in the tens and ones place depending on the inventory value. See above. This is initialized in the main function.

main(): This is the main function in the program. It starts by initializing the various variables needed which can be seen in the code. The function then runs the audio_main function to initially configure the SGTL to allow for audio. After that the program begins loading all the sprites for the characters. This is seen below:

```
for(int row=0; row<16; row++){
    for(int col=0; col<16; col++){
        r = dirt3.pixel_data[3*col+48*row];
        g = dirt3.pixel_data[3*col+48*row+1];
        b = dirt3.pixel_data[3*col+48*row+2];
        //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
        vga_ctrl->VRAM[0x08000+512+2*col+row*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);
        vga_ctrl->VRAM[0x08000+512+2*col+row*32+1] = (r/16)&0x0F;
    }
}
```

This is done for all 32 block sprites used. The offsets are used for the control signals and the multiples of 512 are used to increment from one starting address to another. The rgb are taken from the sprites.h structs and normalized to fit in 16-bits. Note that the inventory is treated as a block, so the inventory is an overlay on top of a sprite somewhere on the world. See below:

```
for(int row=0; row<8; row++){
    for(int col=0; col<7; col++){
        r = numbers8.pixel_data[3*(col+goldi*7)+210*row];
        g = numbers8.pixel_data[3*(col+goldi*7)+210*row+1];
        b = numbers8.pixel_data[3*(col+goldi*7)+210*row+2];
        //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
        vga_ctrl->VRAM[0x08000+5120+2*(col+1)+(row+1)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);
        vga_ctrl->VRAM[0x08000+5120+2*(col+1)+(row+1)*32+1] = (r/16)&0x0F;

        vga_ctrl->VRAM[0x08000+5120+2*(col+8)+(row+7)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);
        vga_ctrl->VRAM[0x08000+5120+2*(col+8)+(row+7)*32+1] = (r/16)&0x0F;

    }
}
```

After this the inventory is drawn and its location is set according to what the designer wants, see below. The inventory is outlined with a white outline for non-selected blocks, and red with selected blocks.

```
for(int row=0; row<16; row++){
        for(int col=0; col<1; col++){
            vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32] = 0; //((g/16)&0
            vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32+1] = 0xFF;//(r/16

            vga_ctrl->VRAM[0x08000+4096+2*(col+15)+(row)*32] = 0; //((g/16
            vga_ctrl->VRAM[0x08000+4096+2*(col+15)+(row)*32+1] = 0xFF;//(r

        }
    }
    for(int row=0; row<1; row++){
        for(int col=0; col<16; col++){
            vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32] = 0; //((g/16)&0
            vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32+1] = 0xFF;//(r/16

            vga_ctrl->VRAM[0x08000+4096+2*(col)+(row+15)*32] = 0; //((g/16
            vga_ctrl->VRAM[0x08000+4096+2*(col)+(row+15)*32+1] = 0xFF;//(r

        }
    }
    for(int row=0; row<16; row++){
            for(int col=0; col<1; col++){
                vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32] = 0xFF; //((
                vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32+1] = 0xFF;//(

                vga_ctrl->VRAM[0x08000+4608+2*(col+15)+(row)*32] = 0xFF; /
                vga_ctrl->VRAM[0x08000+4608+2*(col+15)+(row)*32+1] = 0xFF;

            }
        }
        for(int row=0; row<1; row++){
            for(int col=0; col<16; col++){
                vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32] = 0xFF; //((
                vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32+1] = 0xFF;//(

                vga_ctrl->VRAM[0x08000+4608+2*(col)+(row+15)*32] = 0xFF; /
                vga_ctrl->VRAM[0x08000+4608+2*(col)+(row+15)*32+1] = 0xFF;

            }
        }
        for(int row=0; row<16; row++){
```

The program then sets the character sprites, similar to how it was set for the block sprites. A reflection “algorithm” is used for the left facing sprites.

```
for(int row = 0; row<32; row++){
    for(int col = 0; col<16; col++){
        r = charstatv2.pixel_data[3*(col)+48*row];
        g = charstatv2.pixel_data[3*(col)+48*row+1];
        b = charstatv2.pixel_data[3*(col)+48*row+2];
        //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
        vga_ctrl->VRAM[0x20000+2*col+(row)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);
        vga_ctrl->VRAM[0x20000+2*col+(row)*32+1] = (r/16)&0x0F;// 0x08;//((g/16)&0
    }
}
```

The program then sets the world by first deleting all blocks, setting the 12th layer to the grass block then setting the inventory blocks in a location, the crafting table in a location, as well as the bedrock layer and other boundary blocks.

```
for(int row = 0; row<30; row++){
        for(int col = 0; col<40; col++){
            vga_ctrl->VRAM[2*col+2*row*40] = 0;
        }
        }

for(int row = 12; row<13; row++){
        for(int col = 0; col<40; col++){
            vga_ctrl->VRAM[2*(col)+2*row*40] = 1;
        }
        }

for(int row = 28; row<29; row++){
        for(int col = 0; col<40; col++){
            vga_ctrl->VRAM[2*(col)+2*row*40] = 5-2*(rand()%3%2); // Randomizes bedrock
        }
        }

for(int row = 29; row<30; row++){
        for(int col = 0; col<40; col++){
            vga_ctrl->VRAM[2*(col)+2*row*40] = 5;
        }
        }

for(int row = 13; row<15; row++){
        for(int col = 0; col<40; col++){
            vga_ctrl->VRAM[2*(col)+2*row*40] = 2;
        }
        }

for(int row = 11; row<12; row++){
        for(int col = 35; col<36; col++){
            vga_ctrl->VRAM[2*(col)+2*row*40] = 11;
        }
        }

for(int row = 1; row<2; row++){
        for(int col = 0; col<40; col++){
            vga_ctrl->VRAM[2*(col)+2*row*40] = 13;
        }
        }
```

The program then randomizes the world by splitting the bottom layers into two, the underground, cavern. The underground is 70% dirt, 1 percent gold, and 29% cobble. The cavern is 1 percent bedrock, 6 percent dirt, 8 percent gold, and 85 percent cobble. This is done by using a rand() function which is modulated to 100%, and then each block is set depending on the output, note that this is pseudo random and a different result can be had by setting a different seed. There are other cosmetic things that do not need to be noted.

```
//////////////////     Setting Background Pallete      ////////////////////////////////////
    for(int p = 0; p<64; p++){
        vga_ctrl->VRAM[0x18000+2*p] = (header_data_cmap[p][1]/16)<<4 | (header_data_cmap[p][2]/16);
        vga_ctrl->VRAM[0x18000+2*p+1] = header_data_cmap[p][0]/16;
    }
```

The program then creates some strings that are used for the instruction to the player. The first string is loaded into the VRAM/IP to be shown to the player.

```
char string1[] = "E TO START GAME, 1,2,3 to change blocks, S to build under you          ";
char string2[] = "Move with WAD, Mine 6 cobble with Arrow Keys then press C at bench    ";
char string3[] = "Mine 6 gold using Arrow Keys then press C                             ";
char string4[] = "Mine all the gold block then press enter at bench                     ";
char string5[] = "Still haven't mined all the gold blocks                               ";
char string6[] = "Congrats, you win, do what you want now, please give a good grade    ";

for(int p = 0; p<100; p++){
    vga_ctrl->VRAM[0x38000+2*p+88] = string1[p];
    vga_ctrl->VRAM[0x38000+2*p+88] = string1[p];
    printf("==%x",(int)string1[p]);
}
```

The program then runs USB_in(), which is a carryover from previous labs, then goes into the game while loop. The loop starts by running USB_run and recording the keycode in a variable called keycode. The keycode is then split into the two different keycode1 and keycode2. The program then reads the current character coordinates and stores them. From here, the program responds to any of the arrow keys, S, E, C, enter, and the number keys.

S Key:
If there is no block under the character and the current selected block has non-zero inventory, build under the character. When the block is built the inventory sprite is updated using the loop as before corresponding to the specific block.

```
under = vga_ctrl->VRAM[2*((x)/16+(((y + 15)/16)*16+16)/16*40)];
printf(" undre = %x ", under);
if(under == 0){
    if(vga_ctrl->VRAM[2*((x)/16+(((y + 15)/16)*16+16-16)/16*40)]!=0 || vga_ctrl->VRAM[2*((x)/16+(((y + 15)/16)*16+16+16)/16*40)]!=0 ||
       vga_ctrl->VRAM[2*((x-16)/16+(((y + 15)/16)*16+16)/16*40)]!=0 || vga_ctrl->VRAM[2*((x+16)/16+(((y + 15)/16)*16+16)/16*40)]!=0)

    switch(inven){
        case    2 :

            if(dirti>0){

                vga_ctrl->VRAM[2*((x)/16+(((y + 15)/16)*16+16)/16*40)]= inven;


                dirti = dirti - 1;

            for(int row=0; row<8; row++){
                for(int col=0; col<7; col++){
                    r = numbers8.pixel_data[3*(col+(dirti)/10*7)+210*row];
                    g = numbers8.pixel_data[3*(col+(dirti)/10*7)+210*row+1];
                    b = numbers8.pixel_data[3*(col+(dirti)/10*7)+210*row+2];
                    //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
                    vga_ctrl->VRAM[0x08000+4096+2*(col+1)+(row+1)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);//0x40;//((b/16)&0x0F)<<4 | ((g/16)
                    vga_ctrl->VRAM[0x08000+4096+2*(col+1)+(row+1)*32+1] = (r/16)&0x0F;

                    r = numbers8.pixel_data[3*(col+((dirti)-(dirti)/10*10)*7)+210*row];
                    g = numbers8.pixel_data[3*(col+((dirti)-(dirti)/10*10)*7)+210*row+1];
                    b = numbers8.pixel_data[3*(col+((dirti)-(dirti)/10*10)*7)+210*row+2];

                    vga_ctrl->VRAM[0x08000+4096+2*(col+8)+(row+7)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);//0x40;//((b/16)&0x0F)<<4 | ((g/16)
                    vga_ctrl->VRAM[0x08000+4096+2*(col+8)+(row+7)*32+1] = (r/16)&0x0F;

                }
            }
            }
            break;
        case    3 :

            if(cobblei>0){
```

Arrow Keys: set block to the address corresponding to the block closest to the character in the direction of the arrow key pressed. Then run the mine function with the inventory, progress, and block address.

```
if((keycode1) == 0x50 || (keycode2) == 0x50){    //// Left Arrow
    if(vga_ctrl->VRAM[2*((x-16)/16+(y-15)/16*40)]!=0){
    //  vga_ctrl->VRAM[2*((x-16)/16+(y-15)/16*40)]=0;
        block = 2*((x-16)/16+(y-15)/16*40);

        mine(&dirti, &cobblei, &goldi, block, prog);
```

E Key: If the E key is pressed, update the string to show the next instruction.

```
if((keycode<<8)>>8 == 0x08 || (keycode>>8) == 0x08){
    for(int p = 0; p<100; p++){
        vga_ctrl->VRAM[0x38000+2*p+88] = string2[p];
        vga_ctrl->VRAM[0x38000+2*p+88] = string2[p];
        printf("==%x",(int)string1[p]);
    }
}
```

1,2,3 Keys: The number keys change the selected block by setting inventory to 2, 3 or 4. The program then changes the outline of each of the inventory sprites to indicate which block is currently selected

```
if((keycode1) == 0x1e || (keycode2)== 0x1e){    //1 Arrow
    inven = 2;

    for(int row=0; row<16; row++){
            for(int col=0; col<1; col++){
                vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32] = 0; //((g/16)&0x0F)<<4 | ((b/16)&(
                vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32+1] = 0xFF;//(r/16)&0x0F;

                vga_ctrl->VRAM[0x08000+4096+2*(col+15)+(row)*32] = 0; //((g/16)&0x0F)<<4 | ((b/1(
                vga_ctrl->VRAM[0x08000+4096+2*(col+15)+(row)*32+1] = 0xFF;//(r/16)&0x0F;

            }
        }

        for(int row=0; row<1; row++){
            for(int col=0; col<16; col++){
                vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32] = 0; //((g/16)&0x0F)<<4 | ((b/16)&(
                vga_ctrl->VRAM[0x08000+4096+2*(col)+(row)*32+1] = 0xFF;//(r/16)&0x0F;

                vga_ctrl->VRAM[0x08000+4096+2*(col)+(row+15)*32] = 0; //((g/16)&0x0F)<<4 | ((b/1(
                vga_ctrl->VRAM[0x08000+4096+2*(col)+(row+15)*32+1] = 0xFF;//(r/16)&0x0F;

            }
        }

        for(int row=0; row<16; row++){
                for(int col=0; col<1; col++){
                    vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32] = 0xFF; //((g/16)&0x0F)<<4 | (
                    vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32+1] = 0xFF;//(r/16)&0x0F;

                    vga_ctrl->VRAM[0x08000+4608+2*(col+15)+(row)*32] = 0xFF; //((g/16)&0x0F)<<4 |
                    vga_ctrl->VRAM[0x08000+4608+2*(col+15)+(row)*32+1] = 0xFF;//(r/16)&0x0F;

                }
            }

            for(int row=0; row<1; row++){
                for(int col=0; col<16; col++){
                    vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32] = 0xFF; //((g/16)&0x0F)<<4 | (
                    vga_ctrl->VRAM[0x08000+4608+2*(col)+(row)*32+1] = 0xFF;//(r/16)&0x0F;

                    vga_ctrl->VRAM[0x08000+4608+2*(col)+(row+15)*32] = 0xFF; //((g/16)&0x0F)<<4 |
                    vga_ctrl->VRAM[0x08000+4608+2*(col)+(row+15)*32+1] = 0xFF;//(r/16)&0x0F;

                }
            }
```

C key: This is the crafting key. If the character next to the crafting table, run the if statements. If the progression = 0, then if you have 6 cobble stones and press C, reduce the cobblestone

inventory by 6 and then redraw the mining sprite to show that a new pickaxe has been acquired. The progression variable prog is then incremented. This is then repeated for prog = 1 and prog = 2.

```
if((keycode1) == 0x06 || (keycode2)== 0x06){    //C Key to craft
    if(vga_ctrl->VRAM[2*((x+16)/16+(((y + 15)/16)*16)/16*40)]==11 || vga_ctrl->VRAM[2*((x-16)/16+(((y + 15)/16)*16)/16*40)]==11){
        if(prog==0){
        if(cobblei>=6){

            cobblei = cobblei - 6;

            prog = 1;

            for(int row=0; row<8; row++){
                for(int col=0; col<7; col++){
                    r = numbers8.pixel_data[3*(col+(cobblei)/10*7)+210*row];
                    g = numbers8.pixel_data[3*(col+(cobblei)/10*7)+210*row+1];
                    b = numbers8.pixel_data[3*(col+(cobblei)/10*7)+210*row+2];
                    //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
                    vga_ctrl->VRAM[0x08000+4608+2*(col+1)+(row+1)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);//0x40;//((b/16)&0x0F)<<
                    vga_ctrl->VRAM[0x08000+4608+2*(col+1)+(row+1)*32+1] = (r/16)&0x0F;

                    r = numbers8.pixel_data[3*(col+((cobblei)-(cobblei)/10*10)*7)+210*row];
                    g = numbers8.pixel_data[3*(col+((cobblei)-(cobblei)/10*10)*7)+210*row+1];
                    b = numbers8.pixel_data[3*(col+((cobblei)-(cobblei)/10*10)*7)+210*row+2];

                    vga_ctrl->VRAM[0x08000+4608+2*(col+8)+(row+7)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);//0x40;//((b/16)&0x0F)<<
                    vga_ctrl->VRAM[0x08000+4608+2*(col+8)+(row+7)*32+1] = (r/16)&0x0F;

                }
            }

            for(int row = 0; row<32; row++){
                for(int col = 0; col<16; col++){
                    r = charhit2.pixel_data[3*(col)+48*row];
                    g = charhit2.pixel_data[3*(col)+48*row+1];
                    b = charhit2.pixel_data[3*(col)+48*row+2];
                    //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
                    vga_ctrl->VRAM[0x20000+6144+2*col+(row)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);//0x40;//((b/16)&0x0F)<<4 | ((
                    vga_ctrl->VRAM[0x20000+6144+2*col+(row)*32+1] = (r/16)&0x0F;// 0x08;//((g/16)&0x0F)<<4 | ((b/16)&0x0F);
                }
            }

            for(int row = 0; row<32; row++){
                for(int col = 0; col<16; col++){
                    r = charhit2.pixel_data[3*(15-col)+48*row];
                    g = charhit2.pixel_data[3*(15-col)+48*row+1];
                    b = charhit2.pixel_data[3*(15-col)+48*row+2];
                    //printf("r=%d g=%d b=%d",r*2/32,g*2/32,b*2/32);
                    vga_ctrl->VRAM[0x20000+6144+7168+2*col+(row)*32] = ((g/16)&0x0F)<<4 | ((b/16)&0x0F);//0x40;//((b/16)&0x0F)<<4
                    vga_ctrl->VRAM[0x20000+6144+7168+2*col+(row)*32+1] = (r/16)&0x0F;// 0x08;//((g/16)&0x0F)<<4 | ((b/16)&0x0F);
```

Enter Key:

If the enter key is pressed the program looks to see if the character is next to the crafting table. If the player is, the program loops across the entire world and reads whether there are any gold or half broken gold blocks. If there is a message saying that there is still gold left, if there is no more gold, a congratulations message is given and then an easter egg is drawn.

```
if((keycode1) == 0x28 || (keycode2)== 0x28){//Enter Key for final thing
if(vga_ctrl->VRAM[2*((x+16)/16+(((y + 15)/16)*16)/16*40)]==11 || vga_ctrl->VRAM[2*((x-16)/16+(((y + 15)/16)*16)/16*40)]==11){

    if(prog==2){
        goldflag = 0;
        for(int row=0; row<30; row++){
            for(int col = 0; col<40; col++){
                if(vga_ctrl->VRAM[2*(col+row*40)] == 4){
                    goldflag=1;
                }
                if(vga_ctrl->VRAM[2*(col+row*40)] == 7){
                    goldflag = 1;
                }
        }
        }
        if(goldflag==1){
            for(int p = 0; p<100; p++){
                vga_ctrl->VRAM[0x38000+2*p+88] = string5[p];
                vga_ctrl->VRAM[0x38000+2*p+88] = string5[p];
            }
        }
        if (goldflag==0){
            for(int p = 0; p<100; p++){
                vga_ctrl->VRAM[0x38000+2*p+88] = string6[p];
                vga_ctrl->VRAM[0x38000+2*p+88] = string6[p];
            }

            for(int row = 8; row<12; row++){
                        for(int col = 1; col<5; col++){
                            vga_ctrl->VRAM[2*(col)+2*row*40] = 15+(row-8)*4+(col-1);
                        }
                        }

        }
    }

    goldflag = 0;

}
}
```

The program then loops back to the USB_run.

Post Lab Questions

1. Refer to the Design Resources and Statistics in IQT.17-19 and complete the following design statistics table

LUT	34,175
DSP	0
Memory Bits	869,164, uses 125 M9Ks
Flip-Flop	5,605
Frequency Mhz	75.52
Static Power mW	97.59

Dynamic Power mW	284.68
Total Power mW	404.73

Conclusion

a. Discuss functionality of your design. If parts of your design didn't work, discuss what could be done to fix it.

Some parts that need improvement are the animation and sprite drawing. This was not really a limitation of the project but a limitation of the photo-editing software used to get the sprites and animations in the first place. It would also be good to optimize the design more so the blocks can be more detailed and include more blocks or objectives. Another avenue of improvement would be to more or less revamp how the audio was working. In its current state, we have all the audio data taking up lots of memory which makes it impossible to scale up and add different sound effects for different aspects of the game. If we wanted to add more sound effects, we could try synthesizing simple sounds instead of relying on .wav files. Another thing that we could have done was lower the volume or figure out a way to stop the normalization of the audio that made all of our audio signals clip.

b. What are some potential extensions of this design?

Some extensions include increasing the size of the world or having multiple levels, increasing the amount of ores, or having different lanes of progression with the game. A cool one-off feature would be to add a 'fog of war' that would shade out what ores are in the ground until you expose it to light. Another good extension could be to have the world move with the character such as in terraria.

Since we have completed creating the backbone of the design, the potential extensions that we could add are only limited to the hardware specs of the FPGA. If given more time, we could have implemented a lot more features that would have improved the game significantly.

Overall, we are very pleased with the final project. We are able to meet all of our requirements and were able to add audio, randomization, progression, as well as animation. The best part is that the "game engine" was designed fully, so the game can easily be expanded by adding new sprites, new blocks, or new worlds which can all be done using C on NIOS.