CSEE W4840 Embedded Systems

The Idaho Trail: A Custom Implementation of the Classic Video Game *The Oregon Trail*

Final Report

Prepared for Professor Stephen A. Edwards

Adam Auer – aha2197

Xingcan "Ricardo" Chen –xc2807

Sunny Qi –sq2284

Columbia University

13 May 2026

Introduction

Motivation

Ever since the first mainstream computer games were developed in the 1970s, the most engaging, thought-provoking, and immersive have (quite arguably) been of the roleplaying genre. Such games invite players to see themselves in the protagonist in a way that others do not: instead of simply controlling the character's movements in predictable static environments, players make choices that affect the game's story, world, and ultimate end state.

We three grew up playing roleplaying games. *The Witcher*, *Mass Effect*, and *Cyberpunk 2077* have hundreds of hours played in our collective Steam libraries, a product of their incredible power to enthrall players through deep character customization, explorable worlds, unique situations, and difficult ethical decisions. We yearned to share our passion for roleplaying games through our Embedded

Systems Project; after hours of research and discussion, we settled on a custom implementation of the timeless classic *The Oregon Trail*.

Why *The Oregon Trail*?

The Oregon Trail's greatest allure was the relative lack of expectation associated with it. When people think of *Pacman*, for example, they have a very rigidly defined vision for the game: a mouthed yellow ball traversing a maze of blue lines, attempting to consume all the white dots on the screen before being killed through contact with a brightly-colored ghost. *The Oregon Trail* has been re-implemented so many times over the years, though, that there is no single popular vision of what an *Oregon Trail* game should be. It began in the 1970s as a text-based game developed for a high school mainframe, was re-implemented in the 1980s with color graphics and expanded gameplay, and was even made into a touch-based game for early smartphones in the 2000s. Free of the constraints of a rigid formula, we took confidence in the idea that we could make the game our own. It could turn out as a rather extravagant project with smooth animations and branching story paths, or it could be a more stripped-down version that demonstrated our prowess with C and Verilog through only a few basic mechanics.

Another advantage we saw in *The Oregon Trail* was its heavy reliance on text. People might have different ideas of the game's graphics and difficulty based on the version they played growing up, but they would all agree that *The Oregon Trail* is a "spreadsheet game." The focus isn't really on the graphics; it's on the numbers expressing the player's chances of success and the humorous situations described in textual narration. We as a group were wary of committing ourselves to a too complicated, graphically-intensive assignment that would require hours of developing artistic sprite designs and the mechanisms to make them move. *The Oregon Trail* was the perfect project for starting simple and adding complexity over time, and it offered us the challenge of rendering several lines of text alongside graphics on the screen, something that other popular projects like *Pacman* and *Minesweeper* simply don't do.

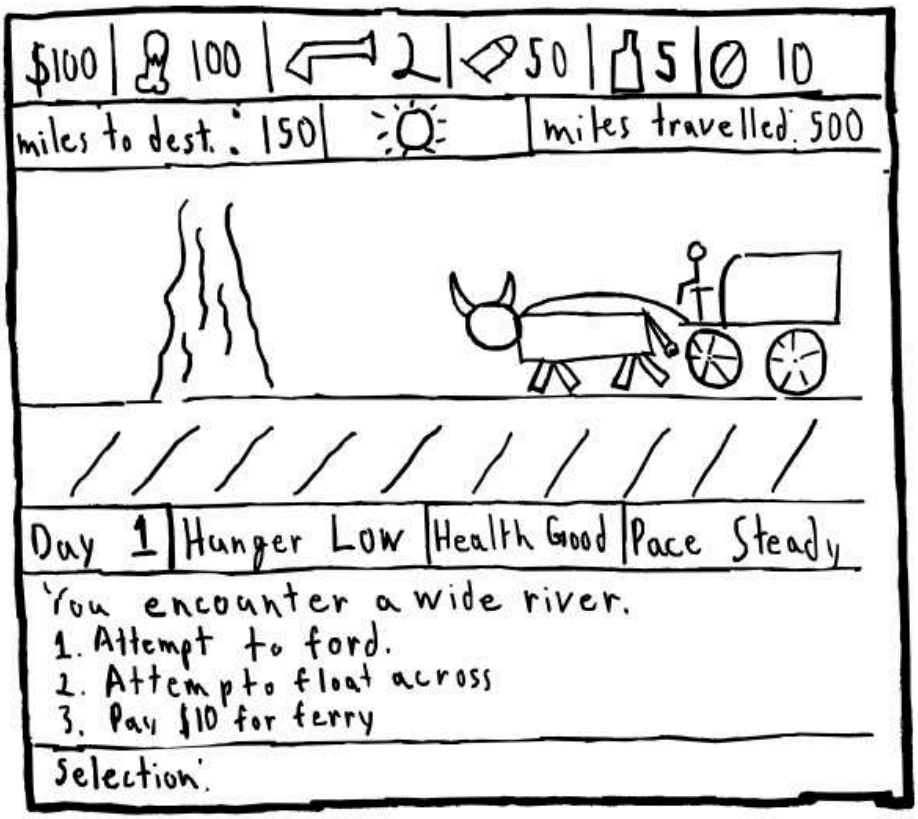


Fig. 1: Early concept sketch

Initial Vision

Our initial vision for the game was rather ambitious. We envisioned the player managing several critical variables, including foodstuffs, guns, ammunition, medicine, spare wheels, weather, and pace, each of which would have different effects on the chance of surviving a multitude of random events. We also envisioned something of a branching side narrative involving a fellow party member making advances on the protagonist's wife, with the player at some point having the option to kill or spare him.

We quickly realized that such a project was far too ambitious for the limited time we had. We pivoted to a much more stripped-down version with only a few stats for the player to manage, a few events, and a few images to excite the player's imagination. Instead of spending hours on game logic and art, we decided that it was more important to focus on developing the custom hardware and software that would allow us to efficiently draw player stats, animated images, narrative text, and a decision prompt to the screen simultaneously.

Big-Picture Block Diagram

Before diving into the details of our implementation, it is useful to begin with a block diagram representing the overall structure of the project.

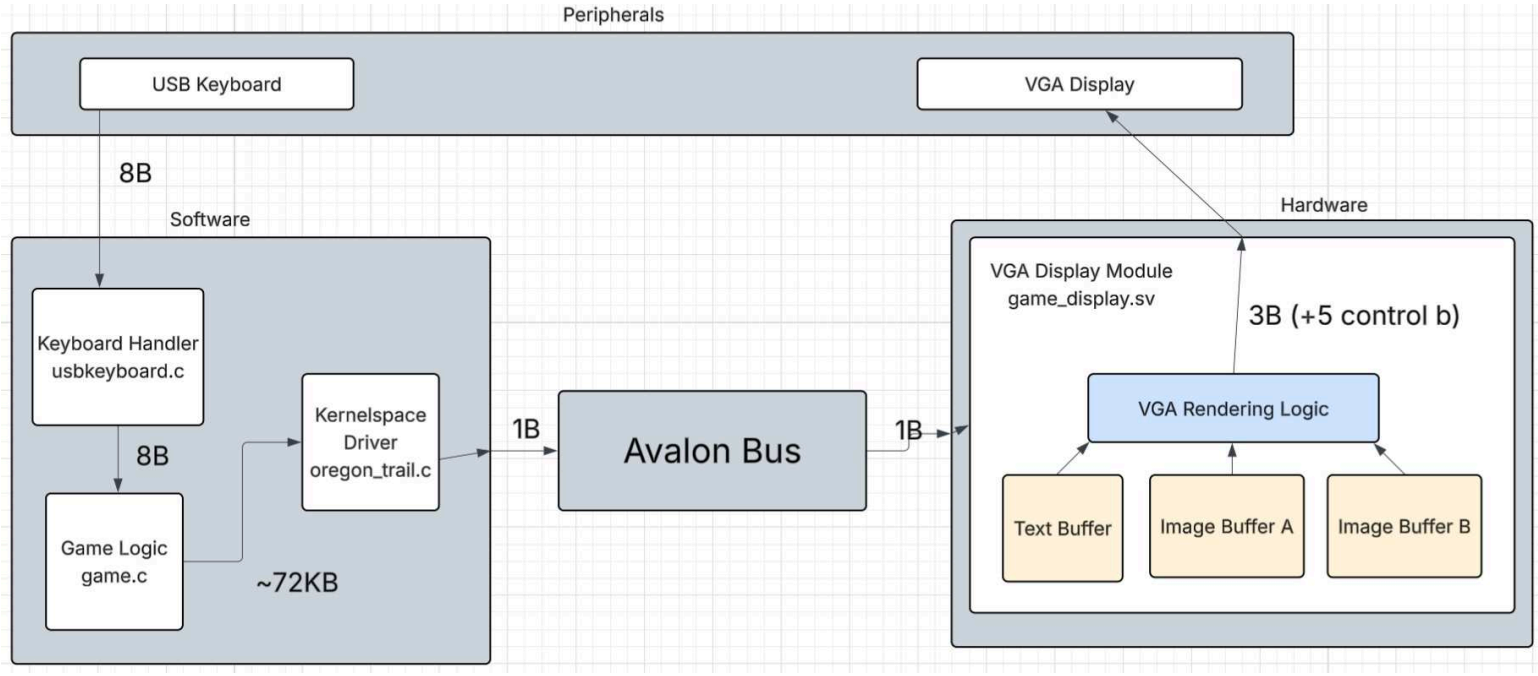


Fig. 2: Big picture block diagram; values represent the size of data passed between components, in bytes (except for the 5 control signals output from VGA Display Module, shown in bits)

Summary

- Peripherals
  - USB keyboard–sole method of user input; for our implementation, only the keys **1-9** are used; all others are ignored
  - VGA display–sole form of output; displays pixels in a 640x480 grid
- Software
  - Keyboard Handler–keyboard inputs are received and interpreted by libusb, managed in userspace by code in usbkeyboard.c; our game logic code in game.c uses usbkeyboard.c/libusb as an intermediary
  - Game Logic–the logic for managing the player's stats and displaying text and images on the screen, written in game.c; interacts with the hardware via the Kernelspace Driver

    - Kernelspace Driver–directly drives the hardware based on data received from the Game Logic; uses iowrite8() to send data over the Avalon bus; writes data into the Text Buffer and both Image Buffers; pixel data is only written to one Image Buffer at a time
- Avalon Bus–transafers data from our Kernelspace Driver to our hardware
- Hardware
    - VGA Display Module–hardware logic for driving the VGA display, described in oregon_trail.c; converts pixel data in the Text Buffer and 8bpp image data in the Image Buffers to the 3 VGA color signals; pixel data is only read from one buffer at a time, whichever one the Kernelspace Driver is not writing to

Display Division

The issue that most radically influenced our later design decisions was the question of how to divide the VGA display into distinct regions for displaying characters and images. We initially devised a rather complicated scheme with different BRAM buffers for the narration box, player prompt, and stat boxes; SDRAM-backed image buffers; and binary to ASCII numeral conversion performed at the hardware level. With the help of Prof. Edwards, though, we developed an ingeniously simple display scheme that greatly simplified our Verilog code.

Physical Layout

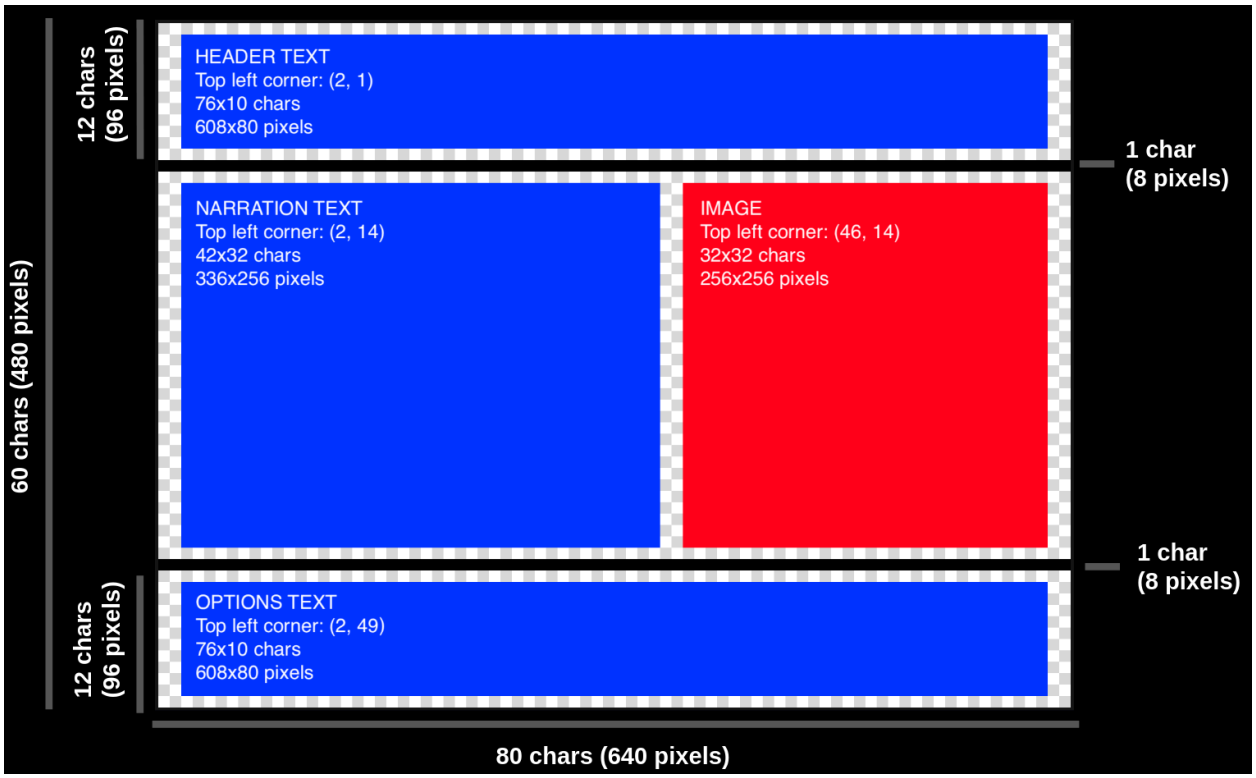


Fig. 3: On-screen layout of our various text and image regions; text regions in BLUE, image region in RED

Here, we show our distinct regions for drawing text and image data as they appear to the player:

- Header Text: displays day number, player health, and player food; extends from one end of the screen to the other
- Narration Text: informs the player of what is currently happening within the game world; extends from the left edge of the screen to the image, which begins at the 39th character column (i.e., where the 49th character in the row would be) and the 385th pixel column
- Image: represents the game world through a hand-drawn image
- Options Text: lists the keys associated with each available player action
- *checkerboard:* always-empty padding rows and columns

Logical Layout (Buffers)

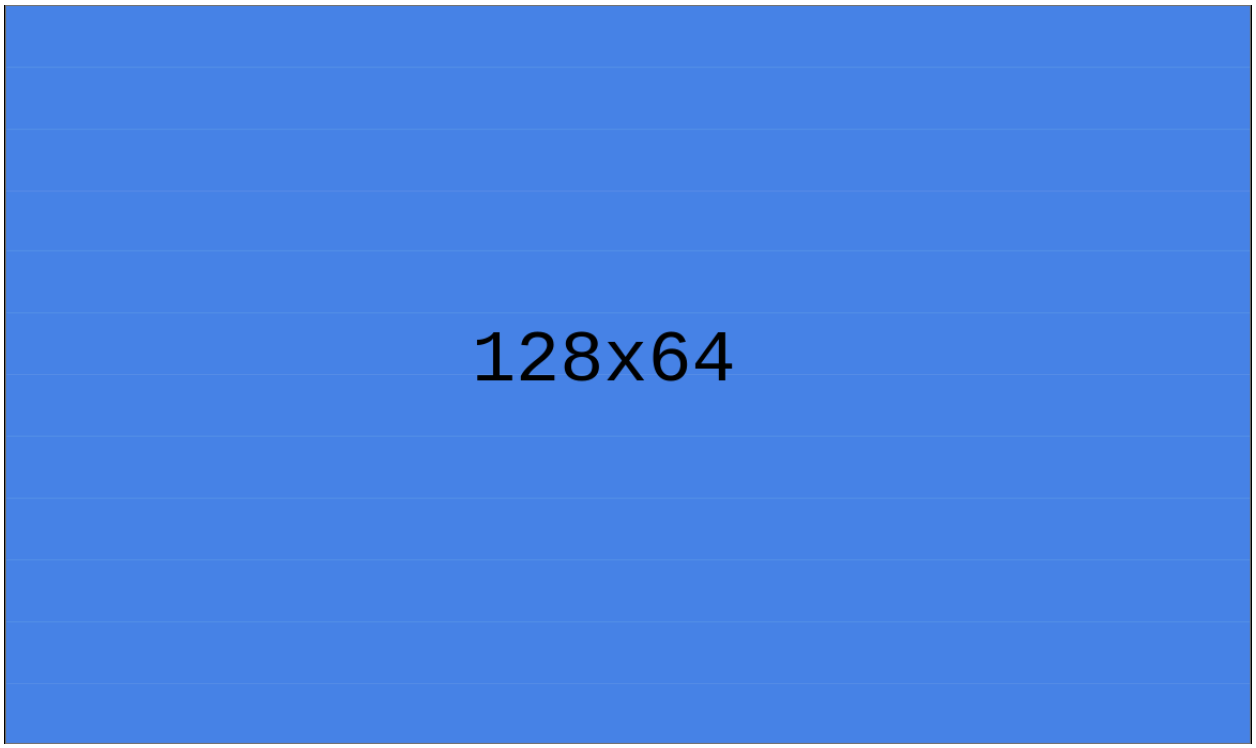


Fig. 4: Text buffer

Fig. 5: Display buffers (A and B)

Here, we demonstrate our scheme for storing text and image data at the hardware level.

Instead of maintaining different buffers, we maintain a single, 8KB array of ASCII characters describing the text for the entire screen. To access a character at a specific *row* and *col* on the screen, we use the formula $arrayIndex = row * NCOLS + col$. Because the number of columns is a power of two, the multiplication and addition simplifies to a bitwise concatenation at the hardware level. Deriving *row* and *col* from *arrayIndex* similarly becomes a bitwise separation of low and high bits. Using an 8x8 font, our display only supports 128 columns and 60 rows of characters, but we simply avoid printing to the "extra" cells or to cells where we know the image to be.

For our image data, we similarly maintain two 64KB buffers, each with exactly enough space to describe a 256x256 (in pixels) image in 8 bit color (RRRGGGBB). Unlike for text, we do use the entire buffer to hold data for the physical screen. These buffers do not cover the entire screen, so we simply take the first column of an image buffer to correspond to the 385th pixel column of the display (with the single exception of the "start" state, when the image is centered). To prevent visible tearing when changing the visible image, our software always writes into the buffer that is not currently displayed.

Buffer Memory Locations

For simplicity, the image and text buffers are allocated in hardware as MK10 memory blocks. The driver can then use iowrite8() to write to effectively draw to the screen directly, avoiding the additional layers of abstraction that would be introduced using SDRAM. Because our images are only 64KB each, the 496KB of BRAM available on the Cyclone V De1-SoC is more than sufficient.

Gameplay Loop

After realizing our initial vision involved far too much coding not directly related to the principles learned in this class, we drastically simplified our gameplay loop to focus on implementing the core elements of *The Oregon Trail* in a manner resistant to edge-cases.

Core Gamestate Structure

Within our userspace C code, the player's states are recorded by the following struct:

```
typedef struct {
  int health;
  int day;
  int money;
  int food;
  int scene;
  int screen_type;
} sw_state_t;
```

Fig. 6: Core player state struct

Intuitively, each integer represents how much a player has of that thing. The one restriction is that each must be representable as an at-most-3-digit, positive number when they are printed, and our game logic simply prevents them from becoming too large or too small. Somewhat counterintuitively, health *increases* as a player's health worsens.

Main Gameplay Loop

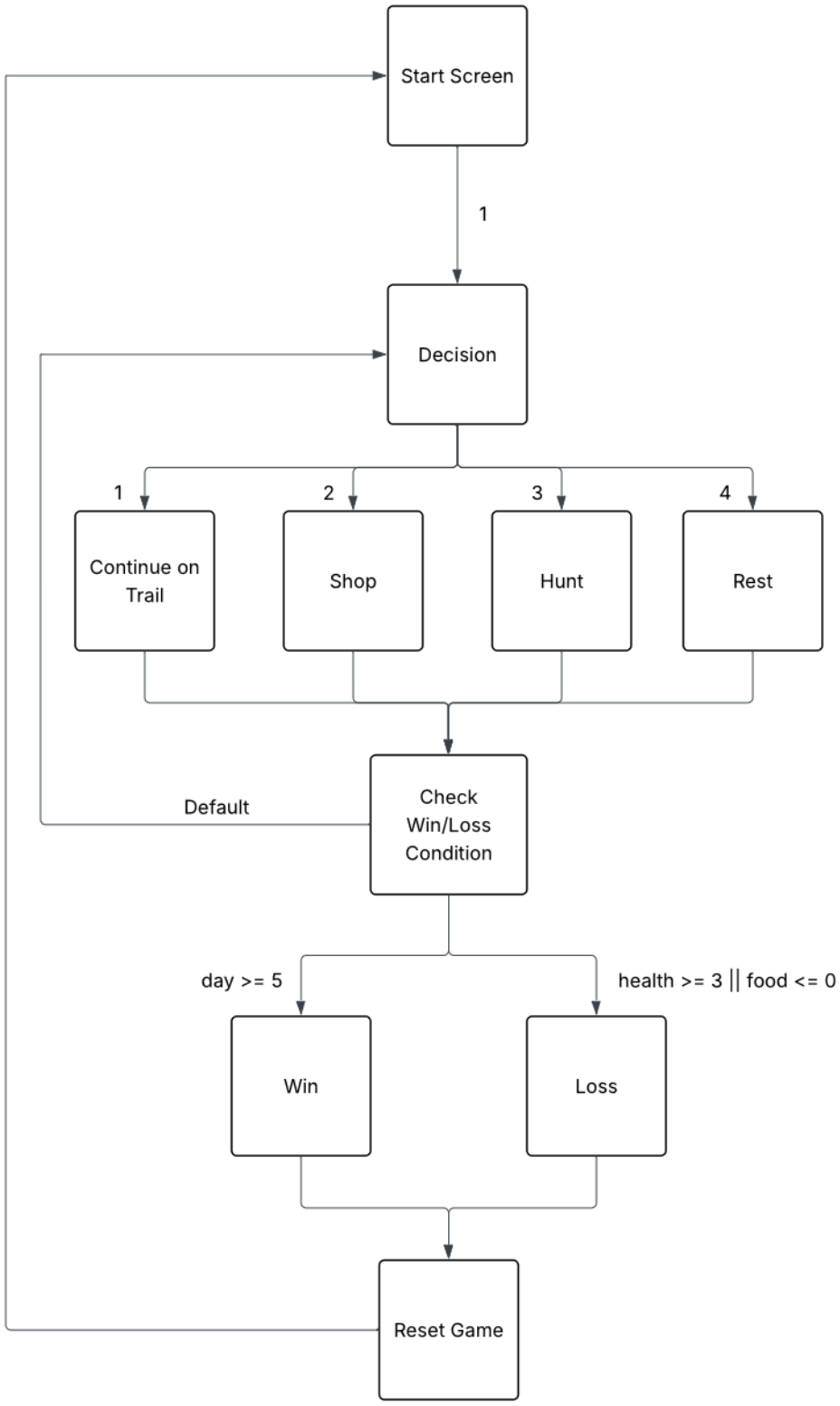


Fig. 7: Gameplay loop diagram

Gameplay State Summary

For all states described, any input other than a valid input is completely ignored. For all states other than the start screen, the image is shown on the right center side of the display.

- Start Screen
    - The start screen appears in the center of the display. The player may press **1** to continue to the first Decision screen.
- Decision
    - The player may press **1**, **2**, **3**, or **4**. A different event then occurs based on the user's input.
    - Show main trail image
- Continue On Trail
    - day += 1
    - food -= 3
    - health += 1
    - show animation
        - exchange two "animation frames" back and forth, three times
- Shop
    - Enter the shop loop
    - User may choose from the following inputs:
        - 1. purchase food
            - 1-9 pounds, at 1 money per pound
        - 2. purchase medicine
            - 1 unit, at 10 money per unit
            - upon purchase, improves health by one degree
        - 3. exit the shop
    - The player may continue to purchase food and medicine until the player elects to exit the shop
    - Show shop image
- Hunt
    - food -= 3
    - generate random 0, 1, or 2
        - 0: fail (with no penalty)
        - 1: food += 5
        - 2: health +=1 (worsening health)

    - Show hunt image
- Rest
    - food -= 3
    - health -= 1 (that is, health *improves* by 1)
    - Show rest image
- Check condition
    - if health >= 3, food <= 0, then loss
    - Otherwise, if day >= 5, then win
- Loss
    - Show loss image
    - After 5 seconds, return to start screen
- Win
    - Show win image
    - After 5 seconds, return to start screen

Reading User Input

For simplicity, we strictly used libusb to read player inputs from a basic USB keyboard. libusb passes data to userspace programs in the following structure:

```
struct usb_keyboard_packet {
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keycode[6];
};
```

Fig. 9: Core structure used by libusb to describe keyboard inputs

Our code in game.c receives this structure, takes keycode[0], and subtracts 0x1D to convert the keypress to one of the digits 0-9 (assuming the key pressed was one of the **1** to **9** keys).

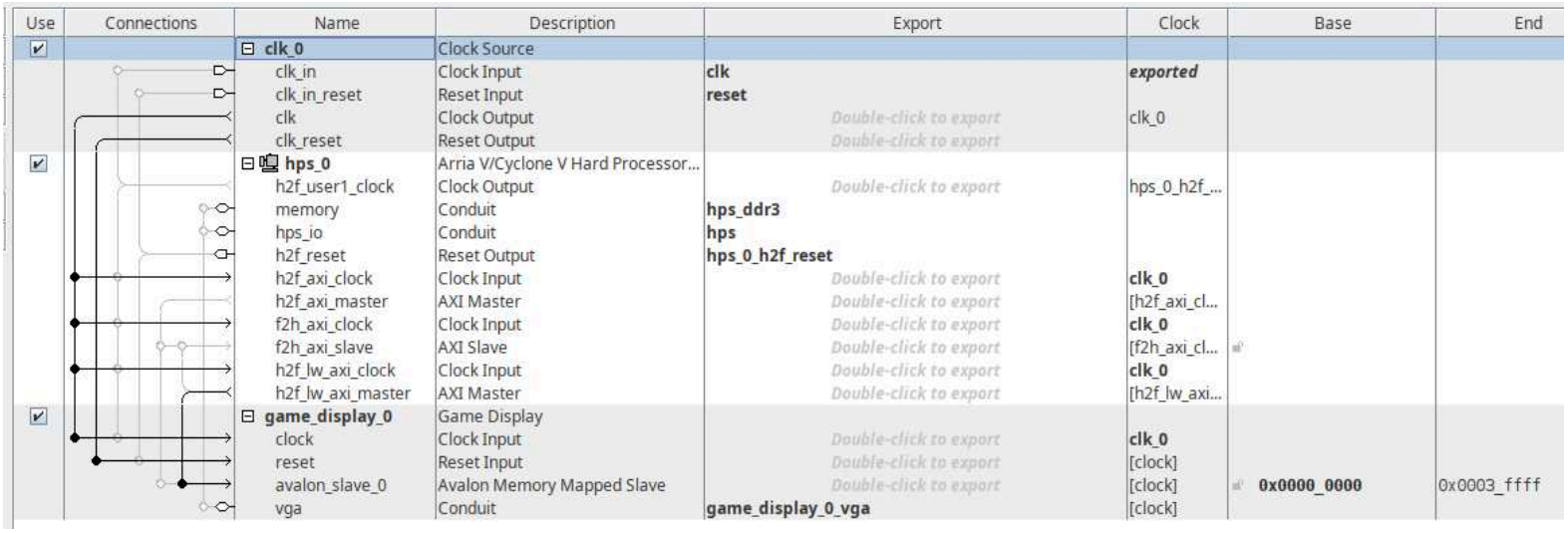
Edge Case Resilience

All invalid inputs at any point in the game are simply ignored. To prevent unexpected behavior resulting from a player holding keys or attempting to enter

multiple inputs simultaneously, we wrap libsub_usb_transfer() in a custom get_key() function. getkey() waits for all keys to be released before returning and returns the integer corresponding to the digit key that was pressed earliest. Keypresses for letters and special characters are ignored. The shift keys are also ignored, so Shift+1, for example, will be interpreted as an input of '1' and not '!'.

Avalon Configuration

To demonstrate how our software and hardware components fit together on the physical Cyclone V De1-Soc, we present screenshots from Intel Platform Designer.



Use	Connections	Name	Description	Export	Clock	Base	End
☑		**clk_0**	Clock Source				
		clk_in	Clock Input	**clk**	***exported***		
		clk_in_reset	Reset Input	**reset**			
		clk	Clock Output	*Double-click to export*	clk_0		
		clk_reset	Reset Output	*Double-click to export*			
☑		**hps_0**	Arria V/Cyclone V Hard Processor...				
		h2f_user1_clock	Clock Output	*Double-click to export*	hps_0_h2f_...		
		memory	Conduit	**hps_ddr3**			
		hps_io	Conduit	**hps**			
		h2f_reset	Reset Output	**hps_0_h2f_reset**			
		h2f_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_cl...		
		f2h_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_cl...		
		h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
		h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi...		
☑		**game_display_0**	Game Display				
		clock	Clock Input	*Double-click to export*	**clk_0**		
		reset	Reset Input	*Double-click to export*	[clock]		
		avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	**0x0000_0000**	0x0003_ffff
		vga	Conduit	**game_display_0_vga**	[clock]		

Fig. 10: Avalon system components

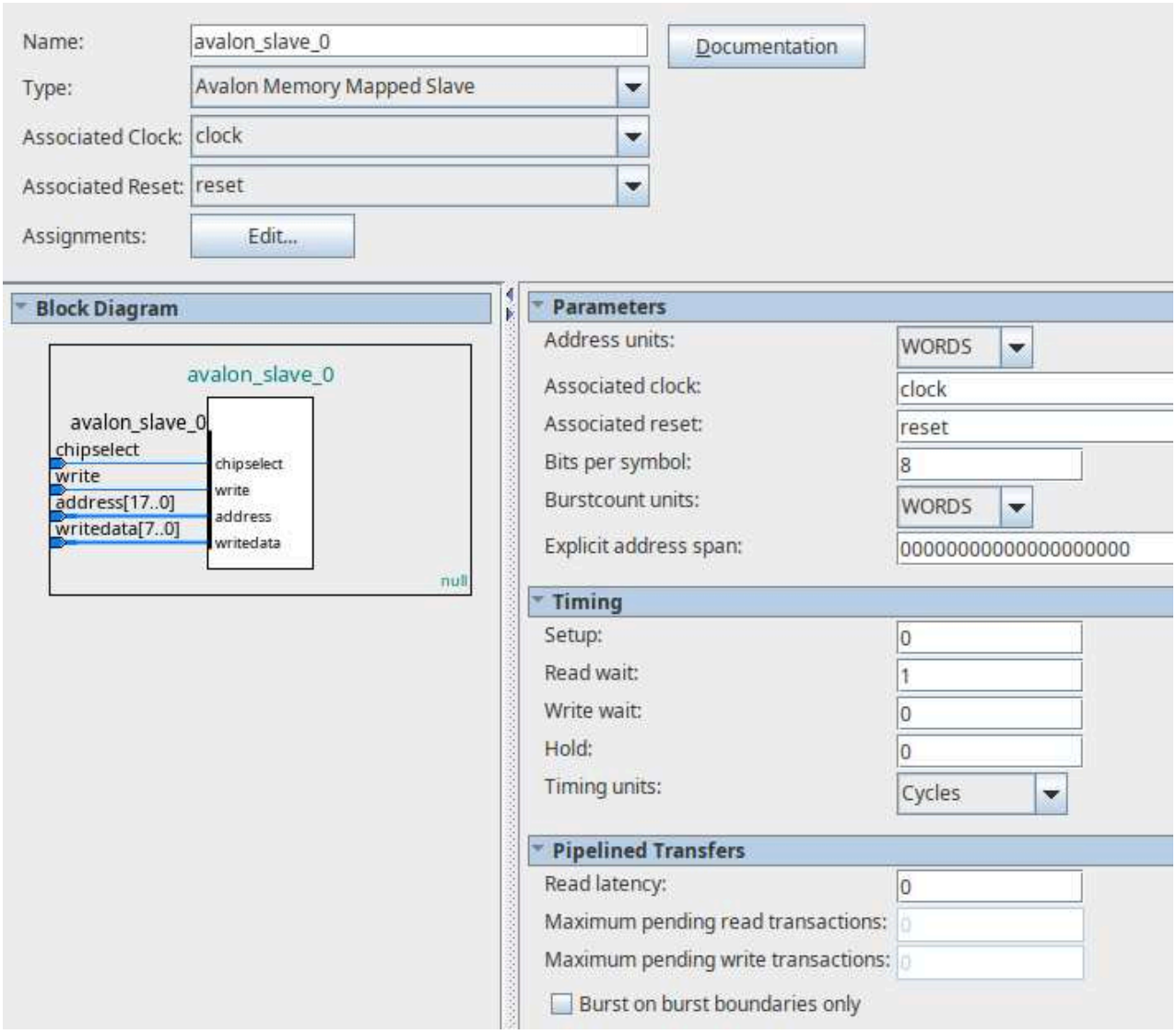


Fig. 11: Software-hardware Avalon interface

System Components

- clk_0
  - provides clock (50 MHz) and reset signals
  - drives the inputs of hps_0 and game_display_0
  - In game_display.sv, we set VGA_CLK to clk, so our VGA output pulses at 50 MHz (it is “double-pumped”)
- hps_0
  - ARM Cortex-A9 hard processor

    - runs our custom C code, including our Game Logic (game.c) and Kernelspace Driver (oregon_trail.c)
- game_display_0
    - custom Avalon memory-mapped slave
        - receives input to the avalon_slave_0 input from hps_0's h2f_lw_axi_master
    - behavior governed by our custom hardware code

Software-Hardware Avalon Interface

avalon_slave_0 serves as the interface between software and hardware, receiving input from hps_0 and setting inputs within game_display_0, an instance of our game_display hardware module defined in game_display.sv. When the Kernel Driver calls iowrite8(), it passes in an address and a single byte of data, which then appear at the address and writedata inputs of avalon_slave. The chipselect and write inputs round out the interface, respectively asserting that the memory-mapped region assigned to the Avalon slave is currently being accessed and that the Avalon slave should read in the current value at the writedata input (which is periodically garbage).

As shown under the "parameters" panel, we chose an 8-bit wide interface for passing data from driver to hardware. Because the driver's primary role is to set 8-bit ASCII characters in the character buffer and 8-bit pixel colors in the image buffers, an 8-bit wide interface is sufficient and simplifies our hardware and software logic. A possible future optimization would be to expand the interface to 16 or 32 bits, allowing us to write multiple pixels or characters simultaneously and avoid the overhead of extra writes over the Avalon Bus.

Register Map

Offset	Size	Name	Type	Access	Desc.
0x00000	1B	CONTROL	boolean	R/W	0 = blank screen 1 = screen on
0x00001	1B	SCREEN_TYPE	boolean	R/W	0 = right-side image 1 = center image

0x00002	1B	ACTIVE_BUF	boolean	R/W	0 = display from buf. A 1 = display from buf. B
0x0003-0x01FFF		*unmapped*			
0x02000-0x03FFF	8KB	TEXT_GRID	M10K BRAM	W	128x64 ASCII character grid
0x04000-0x0FFFF		*unmapped*			
0x10000-0x1FFFF	64KB	IMAGE_A_BUF	M10K BRAM	W	256x256 8bpp image grid
0x20000-0x2FFFF	64KB	IMAGE_B_BUF	M10K BRAM	W	256x256 8bpp image grid

Fig. 12: Register map

Our Kernelspace Driver controls the behavior of game_display_0 by writing to the memory offsets shown in the above register map through avalon_slave_0. By virtue of our simple three-buffer display scheme, our register map is wonderfully concise. Below, we describe the function of each memory region, or register, in some detail:

- CONTROL
    - Controls whether the screen is on or off
    - 0: the values of the R, G, and B signals are set to 0 each clock cycle, making the VGA display black
    - 1: the R, G, and B signals are set based on the values in the Text Buffer and the currently-active Image Buffer
- SCREEN_TYPE
    - 0: the image is displayed at the right-center edge of the VGA display; used throughout the entire game, except for the start screen
    - 1: the image is displayed in the center of the VGA display; used only for the start screen
- ACTIVE_BUF
    - The lynchpin of our double-buffered image display scheme
    - 0: R, G, and B signals are set based on the data in Image Buffer A
        - Kernelspace driver should write to Image Buffer B

    - 1: R, G, and B signals are set based on the data in Image Buffer B
        - Kernelspace driver should write to Image Buffer A
- TEXT_GRID
    - Our text buffer, allocated as seven M10K BRAM blocks
    - Represents 128x64 ASCII character grid (1 character = 1 byte)
    - Given 0-indexed *row* and *col*, the index for a character in the grid is:
        - $arrayIndex = row * 128 + col$
- IMAGE_A_BUF
    - Our first image buffer, allocated as fifty-two M10K BRAM blocks
    - Represents a 256x256 grid of bytes, where each byte describes the color of the corresponding pixel as RRRGGGBB
- IMAGE_B_BUF
    - Our first image buffer, allocated as fifty-two M10K BRAM blocks
    - Represents a 256x256 grid of bytes, where each byte describes the color of the corresponding pixel as RRRGGGBB

Memory Inventory

	Size	Location	Desc.
Font	1 KB (128*8*1)	BRAM (1 block)	Loaded at synthesis time; used to draw characters
Image Buffer A	64 KB (256*256)	BRAM (52 blocks)	Holds image for display or write
Image Buffer B	64 KB (256*256)	BRAM (52 blocks)	Holds image for display or write
Text Buffer	8 KB (128 * 64)	BRAM (7 blocks)	Holds text data; simultaneous read/write
Image Data (.hex)	192 KB	HPS Memory	Loaded on-demand by software

Fig. 13: Memory inventory

In Fig. 13, we present our most significant memory allocations within on-FPGA and on-HPS memory. On the FPGA, the Image Buffers are by far the largest memory-users, taking up 64 KB each. At the beginning of this project, we considered making the displayed image fill the entire width of the screen and half the height, or 640x240 pixels, and representing the images in 16-bit color. Each image buffer would have then taken up 300KB, and the two buffers would have used 600KB of memory total. The Cyclone V De1-SoC comes with only 496KB or BRAM, so our only recourse would have been to place our image buffers in SDRAM, which would have added significant complexity and computational overhead. We thus scaled our images down to 256x256 and switched to 8-bit color, leaving us with a plentiful amount of additional memory should we have needed it later.

Not shown on the Register Map but nonetheless allocated in BRAM is our font data, which describes the shape of each glyph. Our font contains 128 glyphs corresponding to the 128 ASCII characters, and each glyph is 8x8 pixels. For each

glyph, the font contains 8 rows of 8 bits, with a 1 indicating space filled by the drawn character. The full font is thus 128 glyphs * 8 bytes/glyph, or 1KB. At synthesis time, the entire font is loaded from disk by a readmemh command and compiled into the output .sof file. During FPGA configuration, the font is loaded into BRAM, where it can be quickly accessed by our custom hardware.

One final point of interest is that the largest demand our Game Logic places on HPS memory is to hold a single 192KB .hex file describing an image being loaded for display. Instead of converting images from .png to raw binary 8bpp values at runtime, which would waste computational resources unnecessarily, we store images on our SD card in .hex format, where each line of two hexadecimal characters describes a single pixel's color. The hex characters are loaded from disk and parsed into binary whenever an image is needed at runtime, which is only modestly expensive. A potential optimization would be to parse each image into binary pixel data and store it in memory at the start of the Game Logic program (game.c), but our displayed image is updated infrequently enough that this is unnecessary. Converting images to binary pixel data ahead-of-time and saving the resulting .bin files to disk would eliminate the instructions needed for the just-in-time parse step, but keeping images on disk in .hex eases interpretation by humans and LLM's and thus accelerates debugging.

Drawing Text to the Screen

The display controller renders text using a character-based text buffer together with a font ROM. Instead of storing individual text pixels, the software writes ASCII character codes into a dedicated text memory region through the Avalon memory-mapped interface. This greatly reduces memory usage compared to storing full bitmap text images. As mentioned previously, the entire font file is loaded into the .sof file at synthesis time via a single readmemh() call in game_display.sv.

The text buffer is implemented using BRAM. Each memory entry stores one 8-bit ASCII character. The display hardware continuously scans the text buffer while generating VGA output. The text memory is organized as rows of 128 characters per line, although only the first 80 columns are actively displayed. This simplifies address generation because the character row address can be computed using a left shift operation rather than multiplication.

For every visible VGA pixel, the hardware first converts the current VGA scan position into a logical 640×480 coordinate system. The horizontal scan coordinate is divided by two because the VGA timing generator operates at 1280 horizontal pixels while the game internally uses a 640 pixel wide coordinate space.

The corresponding ASCII character is read from text BRAM. The hardware then combines the ASCII value with the current vertical pixel offset inside the character cell to form an address into the font ROM. The font ROM stores one byte per character row, where each bit represents whether a pixel should be illuminated.

The font data is read through a small pipeline. The selected font bit corresponding to the current horizontal offset determines whether the current pixel belongs to the foreground character.

If the font bit is active, the display controller outputs a white pixel. Otherwise, the pixel is treated as transparent, allowing the image layer behind the text to remain visible. Space characters are also treated as transparent to simplify overlay rendering.

Drawing Images to the Screen

Images were initially generated as .png files using the image-creation tools available on pixilart.com. All images but one, the “shop” image displayed during the shop loop, were custom-made. Credit to the shop image (market.png) goes to pixilart user @DgGamez.

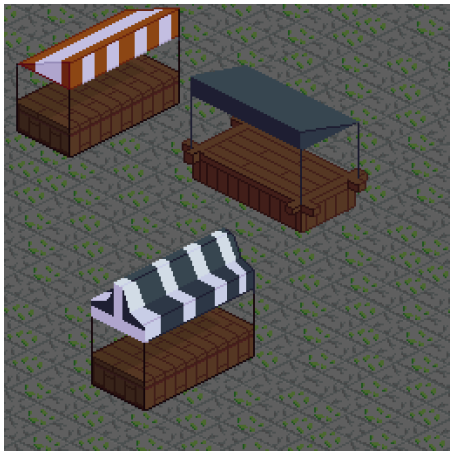


Fig. 14: market.png; credit to @DgGamez

Images were then converted to a .hex file using our png_to_hex.py script. Each line of the output hex file contains the 8-bit (RRRGGGBB) color data for one pixel of the final 256x256 pixel image.

The software writes pixel data into the inactive BRAM buffer through the Avalon memory interface. After the new image is fully written, the software switches the active buffer by updating the active_buf.
During VGA scanout, the display controller converts the current screen position into local image coordinates and calculates the image memory address:

$$imageAddr = y * 256 + x$$

The hardware supports two display modes. In centered-image mode, the image is drawn in the middle of the screen(the size is still 256*256, what has been changed is the position). In gameplay mode, the image is drawn on the right side while the left side is used for text and status information.

The selected image pixel is read from the active frame buffer and expanded into VGA RGB signals. Text pixels have priority over image pixels. Pixels outside the visible display region are forced to black.

Lessons Learned

- Carefully managing memory usage to avoid unexpectedly overusing BRAM
- How userspace code, kernelspace code, and hardware connect through calls to ioctl() and iowriteN()
- Properly communicate with the hardware using the register map
    - The register map should be one of the first things to finish!
- Do it the easy way: instead of writing into many small BRAM buffers, could we instead write into only a few large buffers
- Generating VGA signals in System Verilog based on multiple, potentially overlapping sources of pixel data
- Using LIBUSB to read key presses
    - Address various input bugs
        - Pressing multiple buttons at the same time - forcing only the first button to actually go through
        - Preventing keypresses during sleeps to affect later input (flush keyboard before you receive a new input)

- Carefully consider the sort of physical hardware that the System Verilog code will generate; avoid unnecessary multiplication and division by arbitrary integers; aim for powers of two
- Focus on delivering a "minimum viable product," then try to improve upon that; don't be too ambitious at the start

Individual Contributions

- Sunny Li
    - Basically all software-sided code
    - Implemented entire SW side ioctl logic
        - Size and offsets of all the registers to be sent
    - Handled keyboard input reading and debugging
        - Fixed button mashing but (only first key pressed is registered)
        - Fixed bug of old input (during sleeps) being read for next command (inputs are flushed upon every new decision)
    - Designed and detailed the specs for the 80x60 character grid
    - Ensured all the text inputs remained inside the proper defined bounds
    - Designed image loading logic on SW side
    - All game logic in C
- Ricardo Chen
    - Wrote code for game display hardware
    - Designed VGA display pipeline and framebuffer logic
    - Integrated FPGA hardware with Linux software
    - Debugged VGA timing, framebuffer addressing, and synchronization issues
    - Handled Quartus compile flow and final hardware demo integration
- Adam Auer
    - Drafted initial vision for the overall project
    - Drew and curated images for the game
    - Wrote png_to_hex.py script
    - Researched how to implement image double-buffering with BRAM and SDRAM
    - Helped Sunny and Ricardo debug their software and hardware code
    - Documented the team's interfaces, memory usage, overall design, etc.

- Wrote the majority of the presentation
- Wrote the majority of this report

FMax timing summary

Static timing analysis was performed in Quartus Prime using the Slow 1100mV 85C timing model after full system integration.
The final design achieved:

- FMax: 99.19 MHz
- Restricted FMax: 99.19 MHz
- Clock domain:clock_50_1

Because the target system clock frequency was 50 MHz, the final implementation satisfied all timing constraints with significant timing margin remaining.

Flow Summary

Flow Summary	
Flow Status	Successful - Fri May 8 19:31:32 2026
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	soc_system
Top-level Entity Name	soc_system_top
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	432 / 32,070 (1 %)
Total registers	606
Total pins	362 / 457 (79 %)
Total virtual pins	0
Total block memory bits	1,122,304 / 4,065,280 (28 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	1 / 4 (25 %)

Fig. 14: Flow summary for our final generated hardware logic

Code

```
ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := oregon_trail.o

else

# We are being compiled as a module: use the Kernel build system

        KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
        PWD := $(shell pwd)

LDFLAGS += -lusb-1.0

default: module game

module:
        ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

game: game.c usbkeyboard.c usbkeyboard.h
        ${CC} -o game game.c usbkeyboard.c ${LDFLAGS}

clean:
        ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
        ${RM} game

TARFILES = Makefile README oregon_trail.h oregon_trail.c game.c usbkeyboard.c usbkeyboard.h
TARFILE = lab3-sw.tar.gz
.PHONY : tar
tar : $(TARFILE)

$(TARFILE) : $(TARFILES)
        tar zcfC $(TARFILE) .. $(TARFILES:%=lab3-sw/%)

endif
```

```
 1  #ifndef _OREGON_TRAIL_H
 2  #define _OREGON_TRAIL_H
 3
 4  #include <linux/ioctl.h>
 5  #include <linux/types.h>
 6
 7  #define TEXT_COLS_LOGICAL 128
 8  #define TEXT_ROWS_LOGICAL 64
 9  #define TEXT_MEM_SIZE     (TEXT_COLS_LOGICAL * TEXT_ROWS_LOGICAL)
10
11  #define IMG_SIDE_LOGICAL  256
12  #define IMG_MEM_SIZE      (IMG_SIDE_LOGICAL * IMG_SIDE_LOGICAL)
13
14  #define OFFSET_CONTROL     0x0000  // Screen on/off
15  #define OFFSET_SCREEN_TYPE 0x0001  // Main menu/gameplay screen
16  #define OFFSET_ACTIVE_BUF  0x0002  // Choose Image BRAM dual buffer
17  #define OFFSET_TEXT        0x2000  // 64 rows * 128 cols * 1B = 8kB of text
18  #define OFFSET_IMG_A       0x10000  // 256 * 256 pixels * 1B = 64kB
19  #define OFFSET_IMG_B       0x20000
20
21  #define REG_CONTROL(x)       ((x) + OFFSET_CONTROL)
22  #define REG_SCREEN_TYPE(x)    ((x) + OFFSET_SCREEN_TYPE)
23  #define REG_ACTIVE_BUFFER(x)  ((x) + OFFSET_ACTIVE_BUF)
24  #define REG_TEXT(x)        ((x) + OFFSET_TEXT)
25  #define REG_IMG_A(x)         ((x) + OFFSET_IMG_A)
26  #define REG_IMG_B(x)         ((x) + OFFSET_IMG_B)
27
28
29  typedef struct
30  {
31    __u8  control;       // 0 = Off, 1 = On
32    __u8  screen_type;   // 0 = Gameplay, 1 = Fullscreen
33    __u8  active_buffer; // 0 = Buffer A is active, 1 = Buffer B is active
34
35    char  text_grid[TEXT_MEM_SIZE];
36    __u8  image_data[IMG_MEM_SIZE];
37  } game_state_t;
38
39  typedef struct
40  {
41    game_state_t state;
42  } game_arg_t;
43
44  #define GAME_MAGIC 'q'
45
46  /* ioctls and their arguments */
47  #define GAME_WRITE_STATE _IO(GAME_MAGIC, 1) //try this again
48
49  #endif
```

```c
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod oregon_trail.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree oregon_trail.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "oregon_trail.h"

#define DRIVER_NAME "oregon_trail"

/*
 * Information about our device
 */
struct oregon_trail_dev
{
        struct resource res;    /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
        game_state_t state;
} dev;


/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_game_state(game_state_t *state)
{
        void __iomem *text_ptr;
        void __iomem *img_ptr;
        int i;

```

```
 59          iowrite8(state->control, REG_CONTROL(dev.virtbase));
 60      iowrite8(state->screen_type, REG_SCREEN_TYPE(dev.virtbase));
 61
 62          text_ptr = REG_TEXT(dev.virtbase);
 63
 64      if (state->active_buffer == 0) {
 65          img_ptr = REG_IMG_A(dev.virtbase);
 66      } else {
 67          img_ptr = REG_IMG_B(dev.virtbase);
 68      }
 69
 70          for (i = 0; i < TEXT_MEM_SIZE; i++) {
 71                  iowrite8(state->text_grid[i], text_ptr + i);
 72          }
 73
 74          for (i = 0; i < IMG_MEM_SIZE; i++) {
 75                  iowrite8(state->image_data[i], img_ptr + i);
 76          }
 77
 78          iowrite8(state->active_buffer, REG_ACTIVE_BUFFER(dev.virtbase));
 79
 80          dev.state = *state;
 81  }
 82
 83  /*
 84   * Handle ioctl() calls from userspace:
 85   * Read or write the segments on single digits.
 86   * Note extensive error checking of arguments
 87   */
 88  static long oregon_trail_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
 89  {
 90          game_arg_t *vla;
 91
 92          switch (cmd) {
 93          case GAME_WRITE_STATE:
 94                  //allocate to heap first to avoid stack overflow
 95                  vla = kmalloc(sizeof(game_arg_t), GFP_KERNEL);
 96                  if (!vla) {
 97                          return -ENOMEM;
 98                  }
 99
100                  // Copy to kernel space first to avoid security issues
101                  if (copy_from_user(vla, (game_arg_t *)arg, sizeof(game_arg_t))){
102                          kfree(vla);
103                          return -EACCES;
104                  }
105
106                  write_game_state(&vla->state);
107                  kfree(vla);
108                  break;
109
110          default:
111                  return -EINVAL;
112          }
113          return 0;
114  }
115
116  /* The operations our device knows how to do */
```

```
117  static const struct file_operations oregon_trail_fops = {
118          .owner = THIS_MODULE,
119          .unlocked_ioctl = oregon_trail_ioctl,
120  };
121
122  /* Information about our device for the "misc" framework -- like a char dev */
123  static struct miscdevice oregon_trail_misc_device = {
124          .minor = MISC_DYNAMIC_MINOR,
125          .name = DRIVER_NAME,
126          .fops = &oregon_trail_fops,
127  };
128
129  /*
130   * Initialization code: get resources (registers) and display
131   * a welcome message
132   */
133  static int __init oregon_trail_probe(struct platform_device *pdev)
134  {
135          int ret;
136
137          // obtain hardware register addr
138          ret = misc_register(&oregon_trail_misc_device);
139          if (ret)
140                  return ret;
141
142          /* Get the address of our registers from the device tree */
143          ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
144          if (ret)
145          {
146                  ret = -ENOENT;
147                  goto out_deregister;
148          }
149
150          /* Make sure we can use these registers */
151          // Reserve memory
152          if (request_mem_region(dev.res.start, resource_size(&dev.res),
153                                                  DRIVER_NAME) == NULL)
154          {
155                  ret = -EBUSY;
156                  goto out_deregister;
157          }
158
159          // Map physical addr to kernel's vaddr space
160          dev.virtbase = of_iomap(pdev->dev.of_node, 0);
161      if (dev.virtbase == NULL) {
162          ret = -ENOMEM;
163          goto out_release_mem_region;
164      }
165
166          // Initialize state and send to FPGA
167          memset(&dev.state, 0, sizeof(game_state_t));
168
169      dev.state.control = 1;       // Screen on
170      dev.state.screen_type = 0;   // Gameplay Mode
171      dev.state.active_buffer = 0; // Display Buffer A
172
173      // Fill text grid with spaces first
174      memset(dev.state.text_grid, 0x20, TEXT_MEM_SIZE);
```

```
175
176     write_game_state(&dev.state);
177         return 0;
178
179 out_release_mem_region:
180         release_mem_region(dev.res.start, resource_size(&dev.res));
181 out_deregister:
182         misc_deregister(&oregon_trail_misc_device);
183         return ret;
184 }
185
186 /* Clean-up code: release resources */
187 static int oregon_trail_remove(struct platform_device *pdev)
188 {
189         iounmap(dev.virtbase);
190         release_mem_region(dev.res.start, resource_size(&dev.res));
191         misc_deregister(&oregon_trail_misc_device);
192         return 0;
193 }
194
195 /* Which "compatible" string(s) to search for in the Device Tree */
196 #ifdef CONFIG_OF
197 static const struct of_device_id oregon_trail_of_match[] = {
198         {.compatible = "csee4840,game_display-1.0"},
199         {},
200 };
201 MODULE_DEVICE_TABLE(of, oregon_trail_of_match);
202 #endif
203
204 /* Information for registering ourselves as a "platform" driver */
205 static struct platform_driver oregon_trail_driver = {
206         .driver = {
207                 .name = DRIVER_NAME,
208                 .owner = THIS_MODULE,
209                 .of_match_table = of_match_ptr(oregon_trail_of_match),
210         },
211         .remove = __exit_p(oregon_trail_remove),
212 };
213
214 /* Called when the module is loaded: set things up */
215 static int __init oregon_trail_init(void)
216 {
217         pr_info(DRIVER_NAME ": init\n");
218         return platform_driver_probe(&oregon_trail_driver, oregon_trail_probe);
219 }
220
221 /* Calball when the module is unloaded: release resources */
222 static void __exit oregon_trail_exit(void)
223 {
224         platform_driver_unregister(&oregon_trail_driver);
225         pr_info(DRIVER_NAME ": exit\n");
226 }
227
228 module_init(oregon_trail_init);
229 module_exit(oregon_trail_exit);
230
231 MODULE_LICENSE("GPL");
232 MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
```

```
233  MODULE_DESCRIPTION("VGA ball driver");
```

```c
/*
 * Userspace program that communicates with the oregon_trail device driver
 * through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

#include <stdio.h>
#include "oregon_trail.h"
#include "usbkeyboard.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define SCENE_MAIN  1
#define SCENE_DECISION 2
#define SCENE_TRAIL_ONE 3
#define SCENE_TRAIL_TWO 4
#define SCENE_SHOP  5
#define SCENE_HUNT  6
#define SCENE_REST  7
#define SCENE_LOSE 8
#define SCENE_WIN 9

#define SCENE_SNAKE 10 //UNUSED
#define SCENE_BANDITS 11 //UNUSED

#define SCREEN_COLS 80 // Actual no. of character cols
#define SCREEN_ROWS 60 // Actual no. of character rows

#define HEADER_X 2
#define HEADER_Y 1
#define HEADER_W 76
#define HEADER_H 10

#define TOP_BARRIER_Y 12

#define NARRATION_X 2
#define NARRATION_Y 14
#define NARRATION_W 42
#define NARRATION_H 32

#define BOT_BARRIER_Y 47

#define OPTIONS_X 2
#define OPTIONS_Y 49
#define OPTIONS_W 76
#define OPTIONS_H 10

int oregon_trail_fd;
game_state_t hw_state = {0}; // FPGA tracker

struct libusb_device_handle *keyboard;
```

```c
uint8_t endpoint_address;

typedef struct {
  int health;
  int day;
  int money;
  int food;
  int scene;
  int screen_type;
} sw_state_t;

void set_char(int x, int y, char c);
void draw_h_line(int y, char c);
void print_bounded(int start_x, int start_y, int box_w, int box_h, const char *str);

int load_image_from_hex(const char *filename);
void load_scene_image(int scene);

void update_display(sw_state_t *sw, const char *msg, const char *opts, int sleep_us);
void flush_keyboard();
int get_key();

void reset_game(sw_state_t *sw);
int check_condition(sw_state_t *sw);
void handle_trail(sw_state_t *sw);
void handle_shop(sw_state_t *sw);
void handle_hunt(sw_state_t *sw);
void handle_rest(sw_state_t *sw);

// Set character, ensures it stays within screen bounds
void set_char(int x, int y, char c) {
  // ensure in bounds
  if (x < 0 || x >= SCREEN_COLS || y < 0 || y >= SCREEN_ROWS) return;
  int index = (y * TEXT_COLS_LOGICAL) + x;
  hw_state.text_grid[index] = c;
}

/* Draws a horizontal line across the screen */
void draw_h_line(int y, char c) {
  for (int x = 0; x < SCREEN_COLS; x++) {
    set_char(x, y, c);
  }
}

/* Prints text constrained to a specific bounding box */
void print_bounded(int start_x, int start_y, int box_w, int box_h, const char *str) {
  int x = start_x;
  int y = start_y;

  while (*str) {
    if (*str == '\n') {
      x = start_x;
      y++;
    } else {
      set_char(x, y, *str);
      x++;
      if (x >= start_x + box_w) {
          x = start_x;
```

```
            y++;
        }
      }
      if (y >= start_y + box_h) break;
      str++;
    }
  }

  int load_image_from_hex(const char *filename) {
    char filepath[256];
    snprintf(filepath, sizeof(filepath), "images/%s", filename);

    FILE *fp = fopen(filepath, "r");
    if (!fp) {
      fprintf(stderr, "Could not open image file %s\n", filename);
      return -1;
    }

    for (int i = 0; i < IMG_MEM_SIZE; i++) {
      unsigned int byte;
      if (fscanf(fp, "%2x", &byte) != 1) {
        fprintf(stderr, "Failed to parse byte %d in %s\n", i, filename);
        fclose(fp);
        return -1;
      }
      hw_state.image_data[i] = (__u8)byte;
    }

    fclose(fp);
    return 0;
  }

  void load_scene_image(int scene) {
    static int last_loaded = -1;
    if (scene == last_loaded) return;  // don't reload same image

    const char *filename = NULL;
    switch (scene) {
      case SCENE_MAIN: filename = "start.hex"; break;
      case SCENE_DECISION: filename = "river.hex"; break;
      case SCENE_TRAIL_ONE: filename = "on_the_trail_frame1.hex"; break;
      case SCENE_TRAIL_TWO: filename = "on_the_trail_frame2.hex"; break;
      case SCENE_SHOP: filename = "market.hex"; break;
      case SCENE_HUNT: filename = "hunting.hex"; break;
      case SCENE_REST: filename = "campfire.hex"; break;
      case SCENE_LOSE: filename = "death.hex"; break;
      case SCENE_WIN: filename = "survival.hex"; break;
    }
    // Check for actual file names
    if (filename && load_image_from_hex(filename) == 0) {
        last_loaded = scene;
    }
  }


  void update_display(sw_state_t *sw, const char *msg, const char *opts, int sleep_us) {
    load_scene_image(sw->scene);

```

```
175     int screen_type_changed = (hw_state.screen_type != sw->screen_type);
176     hw_state.screen_type = sw->screen_type;
177
178     // Clear the screen (fill with spaces)
179     memset(hw_state.text_grid, 0x20, TEXT_MEM_SIZE);
180
181     if (sw->screen_type == 0) { // Gameplay mode
182       char header[128];
183       // 'increase health' by 1 (worsens it by a tier)
184       const char* health_str = (sw->health == 0) ? "GOOD" : (sw->health == 1) ? "FAIR" :
    "POOR";
185       snprintf(header, sizeof(header),
186               "Day: %d\nFood: %d lbs\nMoney: $%d\nHealth: %s",
187               sw->day, sw->food, sw->money, health_str);
188
189       // Print the status in the middle of the top section (y=5)
190
191       // Header Box: (2, 1), 76x10
192       print_bounded(HEADER_X, HEADER_Y, HEADER_W, HEADER_H, header);
193
194       // Barrier line at Row 12
195       draw_h_line(12, '-');
196
197       // Narration Box: (2, 14), 42x32. Print only if msg is non-null
198       if (msg) {
199         print_bounded(NARRATION_X, NARRATION_Y, NARRATION_W, NARRATION_H, msg);
200       }
201
202       // Barrier line at Row 47
203       draw_h_line(47, '-');
204
205       // Options Box: (2, 49), 76x10. Print only if opts is non-null
206       if (opts) {
207         print_bounded(OPTIONS_X, OPTIONS_Y, OPTIONS_W, OPTIONS_H, opts);
208       }
209     }
210
211     if (screen_type_changed) {
212       // Blank screen, send full new frame, then re-enable
213       // Prevent previous screen from being carried over to new frame for a split second
214       hw_state.control = 0;
215       hw_state.active_buffer = (hw_state.active_buffer == 0) ? 1 : 0;
216       game_arg_t vla;
217       vla.state = hw_state;
218       ioctl(oregon_trail_fd, GAME_WRITE_STATE, &vla);
219
220       usleep(50000);
221
222       // Re-enable display - everything is already in place
223       hw_state.control = 1;
224       vla.state = hw_state;
225       ioctl(oregon_trail_fd, GAME_WRITE_STATE, &vla);
226     } else {
227       // Normal update
228       hw_state.active_buffer = (hw_state.active_buffer == 0) ? 1 : 0;
229       hw_state.control = 1;
230       game_arg_t vla;
231       vla.state = hw_state;
```

```
232       ioctl(oregon_trail_fd, GAME_WRITE_STATE, &vla);
233     }
234
235     if (sleep_us > 0) {
236       usleep(sleep_us);
237     }
238   }
239
240   void flush_keyboard() {
241     struct usb_keyboard_packet packet;
242     int transferred;
243
244     // Set a short timeout (100ms) so we don't block forever
245     while (1) {
246       int r = libusb_interrupt_transfer(keyboard, endpoint_address,
247         (unsigned char *) &packet, sizeof(packet),
248         &transferred, 100); // 100ms timeout
249
250       // LIBUSB_ERROR_TIMEOUT means no more packets waiting
251       if (r == LIBUSB_ERROR_TIMEOUT) break;
252
253       // Also stop if no key is currently pressed
254       if (transferred == sizeof(packet) && packet.keycode[0] == 0x00) break;
255     }
256   }
257
258   int get_key() {
259
260     struct usb_keyboard_packet packet;
261     int transferred;
262     int result = 0;
263
264     // Ensure keys pressed from previous loading scenes don't affect current one
265     flush_keyboard();
266
267     // Wait for a valid key press
268     while (1) {
269       libusb_interrupt_transfer(keyboard, endpoint_address,
270           (unsigned char *) &packet, sizeof(packet),
271           &transferred, 0);
272
273       if (transferred == sizeof(packet) && packet.keycode[0] != 0x00) {
274         if (packet.keycode[0] >= 0x1E && packet.keycode[0] <= 0x26) {
275           result = packet.keycode[0] - 0x1D;
276           break;  // got a valid key, now wait for release
277         }
278       }
279     }
280
281     // Wait for all keys to be released before returning
282     // 0x00 is keycode for nothing pressed
283     // Ensures first key pressed always goes through
284     while (1) {
285       libusb_interrupt_transfer(keyboard, endpoint_address,
286           (unsigned char *) &packet, sizeof(packet),
287           &transferred, 0);
288
289       if (transferred == sizeof(packet) && packet.keycode[0] == 0x00)
```

```
290           break;  // all keys released
291     }
292
293     return result;
294   }
295
296   void reset_game(sw_state_t *sw) {
297     sw->health = 0;
298     sw->day = 1;
299     sw->money = 50;
300     sw->food = 10;
301     sw->scene = SCENE_MAIN;
302     sw->screen_type = 1; //fullscreen first
303   }
304
305   int main()
306   {
307     static const char filename[] = "/dev/oregon_trail";
308     sw_state_t sw = {0}; // Initialize all to 0
309
310     printf("Starting Idaho Trail Engine...\n");
311
312     if ((oregon_trail_fd = open(filename, O_RDWR)) == -1) {
313       fprintf(stderr, "could not open %s\n", filename);
314       return -1;
315     }
316
317     if ((keyboard = openkeyboard(&endpoint_address)) == NULL) {
318       fprintf(stderr, "Did not find a keyboard\n");
319       close(oregon_trail_fd);
320       return -1;
321     }
322
323     while(1) {
324       reset_game(&sw);
325       update_display(&sw, NULL, NULL, 0);
326       while(get_key() != 1); // Wait for '1' key to start the game
327
328       sw.screen_type = 0; // Switch to gameplay mode
329       sw.scene = SCENE_DECISION;
330       update_display(&sw, "Welcome to Independence. You get ready to embark on a great
    journey.", NULL, 5000000);
331
332       int choice = 0;
333       while (1) { //Game Loop
334
335         if (check_condition(&sw)) {
336           break;
337         }
338
339         update_display(&sw, "What would you like to do?",
340                           "1. Continue on the trail\n2. Buy supplies\n3. Hunt\n4. Rest", 0);
341
342         choice = get_key();
343
344         switch(choice) {
345           case 1: handle_trail(&sw); break;
346           case 2: handle_shop(&sw); break;
```

```
347           case 3: handle_hunt(&sw); break;
348           case 4: handle_rest(&sw); break;
349           default: update_display(&sw, "Invalid choice.", NULL, 2000000); break;
350         }
351       }
352     }
353
354     close(oregon_trail_fd);
355     return 0;
356   }
357
358   int check_condition(sw_state_t *sw) {
359
360     if (sw->health >= 3) {
361       sw->scene = SCENE_LOSE;
362       update_display(sw, "You have succumbed to your injuries.", NULL, 5000000);
363       return 1;
364     }
365     if (sw->food <= 0) {
366       sw->scene = SCENE_LOSE;
367       update_display(sw, "You have succumbed to starvation.", NULL, 5000000);
368       return 1;
369     }
370     if (sw->day >= 5) {
371       sw->scene = SCENE_WIN;
372       update_display(sw, "Congratulations! You have reached Idaho!.", NULL, 5000000);
373       return 1;
374     }
375     return 0; //alive and still on the trail
376   }
377
378
379
380   void handle_trail(sw_state_t *sw) {
381     int i = 0;
382     sw->scene = SCENE_TRAIL_ONE;
383
384     for (i = 0; i < 6; i++) {
385       update_display(sw, "You continue on the trail. The going is tough.", NULL, 500000);
386
387       if (sw->scene == SCENE_TRAIL_ONE) {
388         sw->scene = SCENE_TRAIL_TWO;
389       } else {
390         sw->scene = SCENE_TRAIL_ONE;
391       }
392
393     }
394
395     sw->day += 1;
396     if (sw->food < 3) sw->food = 0;
397     else sw->food -= 3;
398
399     if (sw->health < 3) sw->health += 1;
400
401     return;
402   }
403
404   void handle_shop(sw_state_t *sw) {
```

```
405     sw->scene = SCENE_SHOP; //shop
406
407     while (1) {
408       update_display(sw,
409       "Howdy! Welcome to the shop. See our offerings below.",
410       "1. Food ($1/pound)\n2. Medicine ($10)\n3. Leave the shop",
411       0);
412
413       int choice = get_key();
414
415       switch(choice) {
416         case 1: { //food
417           update_display(sw, "How many pounds of food would you like to buy?", "Choose
      between 1-9 pounds", 0);
418           int pounds_to_buy = get_key();
419           int cost = pounds_to_buy * 1;
420
421           if (pounds_to_buy < 1 || pounds_to_buy > 9) {
422               update_display(sw, "Invalid quantity.", NULL, 2000000);
423           } else if (sw->money >= cost) {
424               sw->money -= cost;
425               sw->food += pounds_to_buy;
426               update_display(sw, "You purchased food.", NULL, 2000000);
427           } else {
428               update_display(sw, "You don't have enough money!", NULL, 2000000);
429           }
430           break;
431         } case 2: { //medicine
432           if (sw->money >= 10) {
433             if(sw->health <= 0){
434               update_display(sw, "You don't need medicine! You're perfectly healthy!", NULL,
      3000000);
435             } else {
436               sw->money -= 10;
437               if (sw->health > 0) sw->health -= 1;
438               update_display(sw, "Thank you for your patronage!", NULL, 3000000);
439             }
440           } else {
441             update_display(sw, "You don't have enough money!", NULL, 3000000);
442           }
443           break;
444         } case 3: { //leave store
445           update_display(sw, "Thank you for stopping by!", NULL, 3000000);
446           return; //go back to base choices
447         } default: {
448             update_display(sw, "Invalid choice. ", NULL, 3000000);
449             break;
450         }
451       }
452     }
453     return;
454   }
455
456   void handle_hunt(sw_state_t *sw) {
457     sw->scene = SCENE_HUNT; //hunting
458     if (sw->food < 3) sw->food = 0;
459     else sw->food -= 3;
460
```

```
461     update_display(sw, "You go out hunting. It's dangerous but you might find food.", NULL,
    3000000);
462
463     int hunt_outcome = rand() % 3; // 0: fail, 1: success, 2: injury
464
465     if (hunt_outcome == 0) {
466       update_display(sw, "You failed to find any food.", NULL, 3000000);
467     } else if (hunt_outcome == 1) {
468       update_display(sw, "Success! You found some food.", NULL, 3000000);
469       sw->food += 5;
470     } else {
471       update_display(sw, "Oh no! You got injured while hunting.", NULL, 3000000);
472       if (sw->health < 3) sw->health += 1;
473     }
474     return;
475   }
476
477   void handle_rest(sw_state_t *sw) {
478     sw->scene = SCENE_REST; //rest
479     if (sw->food < 3) sw->food = 0;
480     else sw->food -= 3;
481
482     if (sw->health > 0) sw->health -= 1;
483
484     update_display(sw, "You rest by the fire. You feel better.", NULL, 3000000);
485
486     return;
487   }
```

```python
import argparse
import sys
from pathlib import Path

from PIL import Image


def rgb_to_rrrgggbb(r: int, g: int, b: int) -> int:
    # Map 0-255 down to the available levels for each channel.
    # Round-to-nearest: multiply, add half-divisor, integer-divide.
    r3 = (r * 7 + 127) // 255  # 0-7
    g3 = (g * 7 + 127) // 255  # 0-7
    b2 = (b * 3 + 127) // 255  # 0-3

    return (r3 << 5) | (g3 << 2) | b2


def convert(png_path: Path, hex_path: Path) -> tuple[int, int]:
    with Image.open(png_path) as img:
        # Flatten any alpha channel against white so transparent regions
        # don't get encoded as black.
        if img.mode in ("RGBA", "LA") or (img.mode == "P" and "transparency" in img.info):
            background = Image.new("RGB", img.size, (0, 0, 0))
            rgba = img.convert("RGBA")
            background.paste(rgba, mask=rgba.split()[-1])
            img = background
        else:
            img = img.convert("RGB")

        width, height = img.size
        pixels = img.getdata()

        with open(hex_path, "w") as f:
            for r, g, b in pixels:
                f.write(f"{rgb_to_rrrgggbb(r, g, b):02X}\n")

    return width, height


def main() -> int:
    parser = argparse.ArgumentParser(
        description="Convert a PNG to a .hex file in RRRGGGBB 8-bit color format."
    )
    parser.add_argument("input", type=Path, help="Path to input PNG file")
    parser.add_argument(
        "-o",
        "--output",
        type=Path,
        default=None,
        help="Path to output .hex file (default: input with .hex extension)",
    )
    args = parser.parse_args()

    if not args.input.is_file():
        print(f"Error: {args.input} is not a file", file=sys.stderr)
        return 1

    output = args.output if args.output is not None else args.input.with_suffix(".hex")
```

```
59
60      width, height = convert(args.input, output)
61      print(f"Wrote {width}x{height} = {width * height} pixels to {output}")
62      return 0
63
64
65  if __name__ == "__main__":
66      sys.exit(main())
```

```
// ====================================================================
// Copyright (c) 2013 by Terasic Technologies Inc.
// ====================================================================
//
// Modified 2019 by Stephen A. Edwards
//
// Permission:
//
//   Terasic grants permission to use and modify this code for use
//   in synthesis for all Terasic Development Boards and Altera
//   Development Kits made by Terasic.  Other use of this code,
//   including the selling ,duplication, or modification of any
//   portion is strictly prohibited.
//
// Disclaimer:
//
//   This VHDL/Verilog or C/C++ source code is intended as a design
//   reference which illustrates how these types of functions can be
//   implemented.  It is the user's responsibility to verify their
//   design for consistency and functionality through the use of
//   formal verification methods.  Terasic provides no warranty
//   regarding the use or functionality of this code.
//
// ====================================================================
//
//  Terasic Technologies Inc

//  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
//
//
//                     web: http://www.terasic.com/
//                     email: support@terasic.com
module soc_system_top(

 ///////// ADC /////////
 inout         ADC_CS_N,
 output        ADC_DIN,
 input         ADC_DOUT,
 output        ADC_SCLK,

 ///////// AUD /////////
 input         AUD_ADCDAT,
 inout         AUD_ADCLRCK,
 inout         AUD_BCLK,
 output        AUD_DACDAT,
 inout         AUD_DACLRCK,
 output        AUD_XCK,

 ///////// CLOCK2 /////////
 input         CLOCK2_50,

 ///////// CLOCK3 /////////
 input         CLOCK3_50,

 ///////// CLOCK4 /////////
 input         CLOCK4_50,

 ///////// CLOCK /////////
```

```
 59   input         CLOCK_50,
 60
 61   ///////// DRAM /////////
 62   output [12:0] DRAM_ADDR,
 63   output [1:0]  DRAM_BA,
 64   output        DRAM_CAS_N,
 65   output        DRAM_CKE,
 66   output        DRAM_CLK,
 67   output        DRAM_CS_N,
 68   inout [15:0]  DRAM_DQ,
 69   output        DRAM_LDQM,
 70   output        DRAM_RAS_N,
 71   output        DRAM_UDQM,
 72   output        DRAM_WE_N,
 73
 74   ///////// FAN /////////
 75   output        FAN_CTRL,
 76
 77   ///////// FPGA /////////
 78   output        FPGA_I2C_SCLK,
 79   inout         FPGA_I2C_SDAT,
 80
 81   ///////// GPIO /////////
 82   inout [35:0]  GPIO_0,
 83   inout [35:0]  GPIO_1,
 84
 85   ///////// HEX0 /////////
 86   output [6:0]  HEX0,
 87
 88   ///////// HEX1 /////////
 89   output [6:0]  HEX1,
 90
 91   ///////// HEX2 /////////
 92   output [6:0]  HEX2,
 93
 94   ///////// HEX3 /////////
 95   output [6:0]  HEX3,
 96
 97   ///////// HEX4 /////////
 98   output [6:0]  HEX4,
 99
100   ///////// HEX5 /////////
101   output [6:0]  HEX5,
102
103   ///////// HPS /////////0avalon_slave
104   inout         HPS_CONV_USB_N,
105   output [14:0] HPS_DDR3_ADDR,
106   output [2:0]  HPS_DDR3_BA,
107   output        HPS_DDR3_CAS_N,
108   output        HPS_DDR3_CKE,
109   output        HPS_DDR3_CK_N,
110   output        HPS_DDR3_CK_P,
111   output        HPS_DDR3_CS_N,
112   output [3:0]  HPS_DDR3_DM,
113   inout [31:0]  HPS_DDR3_DQ,
114   inout [3:0]   HPS_DDR3_DQS_N,
115   inout [3:0]   HPS_DDR3_DQS_P,
116   output        HPS_DDR3_ODT,
```

```
117   output        HPS_DDR3_RAS_N,
118   output        HPS_DDR3_RESET_N,
119   input         HPS_DDR3_RZQ,
120   output        HPS_DDR3_WE_N,
121   output        HPS_ENET_GTX_CLK,
122   inout         HPS_ENET_INT_N,
123   output        HPS_ENET_MDC,
124   inout         HPS_ENET_MDIO,
125   input         HPS_ENET_RX_CLK,
126   input [3:0]   HPS_ENET_RX_DATA,
127   input         HPS_ENET_RX_DV,
128   output [3:0]  HPS_ENET_TX_DATA,
129   output        HPS_ENET_TX_EN,
130   inout         HPS_GSENSOR_INT,
131   inout         HPS_I2C1_SCLK,
132   inout         HPS_I2C1_SDAT,
133   inout         HPS_I2C2_SCLK,
134   inout         HPS_I2C2_SDAT,
135   inout         HPS_I2C_CONTROL,
136   inout         HPS_KEY,
137   inout         HPS_LED,
138   inout         HPS_LTC_GPIO,
139   output        HPS_SD_CLK,
140   inout         HPS_SD_CMD,
141   inout [3:0]   HPS_SD_DATA,
142   output        HPS_SPIM_CLK,
143   input         HPS_SPIM_MISO,
144   output        HPS_SPIM_MOSI,
145   inout         HPS_SPIM_SS,
146   input         HPS_UART_RX,
147   output        HPS_UART_TX,
148   input         HPS_USB_CLKOUT,
149   inout [7:0]   HPS_USB_DATA,
150   input         HPS_USB_DIR,
151   input         HPS_USB_NXT,
152   output        HPS_USB_STP,
153
154   ///////// IRDA /////////
155   input         IRDA_RXD,
156   output        IRDA_TXD,
157
158   ///////// KEY /////////
159   input [3:0]   KEY,
160
161   ///////// LEDR /////////
162   output [9:0]  LEDR,
163
164   ///////// PS2 /////////
165   inout         PS2_CLK,
166   inout         PS2_CLK2,
167   inout         PS2_DAT,
168   inout         PS2_DAT2,
169
170   ///////// SW /////////
171   input [9:0]   SW,
172
173   ///////// TD /////////
174   input         TD_CLK27,
```

```
175   input [7:0]   TD_DATA,
176   input         TD_HS,
177   output        TD_RESET_N,
178   input         TD_VS,
179
180
181   ///////// VGA /////////
182   output [7:0]  VGA_B,
183   output        VGA_BLANK_N,
184   output        VGA_CLK,
185   output [7:0]  VGA_G,
186   output        VGA_HS,
187   output [7:0]  VGA_R,
188   output        VGA_SYNC_N,
189   output        VGA_VS
190  );
191
192     soc_system soc_system0(
193       .clk_clk                       ( CLOCK_50 ),
194       .reset_reset_n                 ( 1'b1 ),
195
196       .hps_ddr3_mem_a                ( HPS_DDR3_ADDR ),
197       .hps_ddr3_mem_ba               ( HPS_DDR3_BA ),
198       .hps_ddr3_mem_ck               ( HPS_DDR3_CK_P ),
199       .hps_ddr3_mem_ck_n             ( HPS_DDR3_CK_N ),
200       .hps_ddr3_mem_cke              ( HPS_DDR3_CKE ),
201       .hps_ddr3_mem_cs_n             ( HPS_DDR3_CS_N ),
202       .hps_ddr3_mem_ras_n            ( HPS_DDR3_RAS_N ),
203       .hps_ddr3_mem_cas_n            ( HPS_DDR3_CAS_N ),
204       .hps_ddr3_mem_we_n             ( HPS_DDR3_WE_N ),
205       .hps_ddr3_mem_reset_n          ( HPS_DDR3_RESET_N ),
206       .hps_ddr3_mem_dq               ( HPS_DDR3_DQ ),
207       .hps_ddr3_mem_dqs              ( HPS_DDR3_DQS_P ),
208       .hps_ddr3_mem_dqs_n            ( HPS_DDR3_DQS_N ),
209       .hps_ddr3_mem_odt              ( HPS_DDR3_ODT ),
210       .hps_ddr3_mem_dm               ( HPS_DDR3_DM ),
211       .hps_ddr3_oct_rzqin            ( HPS_DDR3_RZQ ),
212
213       .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
214       .hps_hps_io_emac1_inst_TXD0   ( HPS_ENET_TX_DATA[0] ),
215       .hps_hps_io_emac1_inst_TXD1   ( HPS_ENET_TX_DATA[1] ),
216       .hps_hps_io_emac1_inst_TXD2   ( HPS_ENET_TX_DATA[2] ),
217       .hps_hps_io_emac1_inst_TXD3   ( HPS_ENET_TX_DATA[3] ),
218       .hps_hps_io_emac1_inst_RXD0   ( HPS_ENET_RX_DATA[0] ),
219       .hps_hps_io_emac1_inst_MDIO   ( HPS_ENET_MDIO  ),
220       .hps_hps_io_emac1_inst_MDC    ( HPS_ENET_MDC   ),
221       .hps_hps_io_emac1_inst_RX_CTL ( HPS_ENET_RX_DV ),
222       .hps_hps_io_emac1_inst_TX_CTL ( HPS_ENET_TX_EN ),
223       .hps_hps_io_emac1_inst_RX_CLK ( HPS_ENET_RX_CLK ),
224       .hps_hps_io_emac1_inst_RXD1   ( HPS_ENET_RX_DATA[1]  ),
225       .hps_hps_io_emac1_inst_RXD2   ( HPS_ENET_RX_DATA[2]  ),
226       .hps_hps_io_emac1_inst_RXD3   ( HPS_ENET_RX_DATA[3]  ),
227
228       .hps_hps_io_sdio_inst_CMD     ( HPS_SD_CMD          ),
229       .hps_hps_io_sdio_inst_D0      ( HPS_SD_DATA[0]      ),
230       .hps_hps_io_sdio_inst_D1      ( HPS_SD_DATA[1]      ),
231       .hps_hps_io_sdio_inst_CLK     ( HPS_SD_CLK          ),
232       .hps_hps_io_sdio_inst_D2      ( HPS_SD_DATA[2]      ),
```

```
233       .hps_hps_io_sdio_inst_D3      ( HPS_SD_DATA[3]      ),
234
235       .hps_hps_io_usb1_inst_D0      ( HPS_USB_DATA[0]     ),
236       .hps_hps_io_usb1_inst_D1      ( HPS_USB_DATA[1]     ),
237       .hps_hps_io_usb1_inst_D2      ( HPS_USB_DATA[2]     ),
238       .hps_hps_io_usb1_inst_D3      ( HPS_USB_DATA[3]     ),
239       .hps_hps_io_usb1_inst_D4      ( HPS_USB_DATA[4]     ),
240       .hps_hps_io_usb1_inst_D5      ( HPS_USB_DATA[5]     ),
241       .hps_hps_io_usb1_inst_D6      ( HPS_USB_DATA[6]     ),
242       .hps_hps_io_usb1_inst_D7      ( HPS_USB_DATA[7]     ),
243       .hps_hps_io_usb1_inst_CLK     ( HPS_USB_CLKOUT      ),
244       .hps_hps_io_usb1_inst_STP     ( HPS_USB_STP         ),
245       .hps_hps_io_usb1_inst_DIR     ( HPS_USB_DIR         ),
246       .hps_hps_io_usb1_inst_NXT     ( HPS_USB_NXT         ),
247
248       .hps_hps_io_spim1_inst_CLK    ( HPS_SPIM_CLK  ),
249       .hps_hps_io_spim1_inst_MOSI   ( HPS_SPIM_MOSI ),
250       .hps_hps_io_spim1_inst_MISO   ( HPS_SPIM_MISO ),
251       .hps_hps_io_spim1_inst_SS0    ( HPS_SPIM_SS   ),
252
253       .hps_hps_io_uart0_inst_RX     ( HPS_UART_RX     ),
254       .hps_hps_io_uart0_inst_TX     ( HPS_UART_TX     ),
255
256       .hps_hps_io_i2c0_inst_SDA     ( HPS_I2C1_SDAT     ),
257       .hps_hps_io_i2c0_inst_SCL     ( HPS_I2C1_SCLK     ),
258
259       .hps_hps_io_i2c1_inst_SDA     ( HPS_I2C2_SDAT     ),
260       .hps_hps_io_i2c1_inst_SCL     ( HPS_I2C2_SCLK     ),
261
262       .hps_hps_io_gpio_inst_GPIO09  ( HPS_CONV_USB_N ),
263       .hps_hps_io_gpio_inst_GPIO35  ( HPS_ENET_INT_N ),
264       .hps_hps_io_gpio_inst_GPIO40  ( HPS_LTC_GPIO ),
265
266       .hps_hps_io_gpio_inst_GPIO48  ( HPS_I2C_CONTROL ),
267       .hps_hps_io_gpio_inst_GPIO53  ( HPS_LED ),
268       .hps_hps_io_gpio_inst_GPIO54  ( HPS_KEY ),
269       .hps_hps_io_gpio_inst_GPIO61  ( HPS_GSENSOR_INT ),
270
271       .game_display_0_vga_b       (VGA_B),
272       .game_display_0_vga_blank_n (VGA_BLANK_N),
273       .game_display_0_vga_clk     (VGA_CLK),
274       .game_display_0_vga_g       (VGA_G),
275       .game_display_0_vga_hs      (VGA_HS),
276       .game_display_0_vga_r       (VGA_R),
277       .game_display_0_vga_sync_n  (VGA_SYNC_N),
278       .game_display_0_vga_vs      (VGA_VS)
279
280    );
281
282     // The following quiet the "no driver" warnings for output
283     // pins and should be removed if you use any of these peripherals
284
285     assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
286     assign ADC_DIN = SW[0];
287     assign ADC_SCLK = SW[0];
288
289     assign AUD_ADCLRCK = SW[1] ? SW[0] : 1'bZ;
290     assign AUD_BCLK = SW[1] ? SW[0] : 1'bZ;
```

```
291     assign AUD_DACDAT = SW[0];
292     assign AUD_DACLRCK = SW[1] ? SW[0] : 1'bZ;
293     assign AUD_XCK = SW[0];
294
295     assign DRAM_ADDR = { 13{ SW[0] } };
296     assign DRAM_BA = { 2{ SW[0] } };
297     assign DRAM_DQ = SW[1] ? { 16{ SW[0] } } : { 16{ 1'bZ } };
298     assign {DRAM_CAS_N, DRAM_CKE, DRAM_CLK, DRAM_CS_N,
299             DRAM_LDQM, DRAM_RAS_N, DRAM_UDQM, DRAM_WE_N} = { 8{SW[0]} };
300
301     assign FAN_CTRL = SW[0];
302
303     assign FPGA_I2C_SCLK = SW[0];
304     assign FPGA_I2C_SDAT = SW[1] ? SW[0] : 1'bZ;
305
306     assign GPIO_0 = SW[1] ? { 36{ SW[0] } } : { 36{ 1'bZ } };
307     assign GPIO_1 = SW[1] ? { 36{ SW[0] } } : { 36{ 1'bZ } };
308
309     assign HEX0 = { 7{ SW[1] } };
310     assign HEX1 = { 7{ SW[2] } };
311     assign HEX2 = { 7{ SW[3] } };
312     assign HEX3 = { 7{ SW[4] } };
313     assign HEX4 = { 7{ SW[5] } };
314     assign HEX5 = { 7{ SW[6] } };
315
316     assign IRDA_TXD = SW[0];
317
318     assign LEDR = { 10{SW[7]} };
319
320     assign PS2_CLK = SW[1] ? SW[0] : 1'bZ;
321     assign PS2_CLK2 = SW[1] ? SW[0] : 1'bZ;
322     assign PS2_DAT = SW[1] ? SW[0] : 1'bZ;
323     assign PS2_DAT2 = SW[1] ? SW[0] : 1'bZ;
324
325     assign TD_RESET_N = SW[0];
326
327     //assign {VGA_R, VGA_G, VGA_B} = { 24{ SW[0] } };
328     //assign {VGA_BLANK_N, VGA_CLK,
329             //VGA_HS, VGA_SYNC_N, VGA_VS} = { 5{ SW[0] } };
330
331
332 endmodule
```

```
 1  module byte_sdp_ram #(
 2      parameter ADDR_WIDTH = 10,
 3      parameter DEPTH = 1024
 4  ) (
 5      input  logic                  clk,
 6      input  logic                  we,
 7      input  logic [ADDR_WIDTH-1:0] waddr,
 8      input  logic [7:0]            wdata,
 9      input  logic [ADDR_WIDTH-1:0] raddr,
10      output logic [7:0]            rdata
11  );
12      (* ramstyle = "M10K" *) logic [7:0] mem [0:DEPTH-1];
13
14      always_ff @(posedge clk) begin
15          if (we) begin
16              mem[waddr] <= wdata;
17          end
18          rdata <= mem[raddr];
19      end
20  endmodule
21
22  module font_sdp_rom (
23      input  logic       clk,
24      input  logic [9:0] raddr,
25      output logic [7:0] rdata
26  );
27      (* ramstyle = "M10K" *) logic [7:0] mem [0:1023];
28
29      initial begin
30          $readmemh("font.hex", mem);
31      end
32
33      always_ff @(posedge clk) begin
34          rdata <= mem[raddr];
35      end
36  endmodule
37
38  module game_display (
39      input  logic        clk,
40      input  logic        reset,
41
42      input  logic        chipselect,
43      input  logic        write,
44      input  logic [17:0] address,
45      input  logic [7:0]  writedata,
46
47      output logic [7:0]  VGA_R,
48      output logic [7:0]  VGA_G,
49      output logic [7:0]  VGA_B,
50      output logic        VGA_HS,
51      output logic        VGA_VS,
52      output logic        VGA_CLK,
53      output logic        VGA_BLANK_n,
54      output logic        VGA_SYNC_n
55  );
56      localparam int H_VISIBLE = 1280;
57      localparam int H_FRONT   = 32;
58      localparam int H_SYNC    = 192;
```

```
 59      localparam int H_BACK    = 96;
 60      localparam int H_TOTAL   = H_VISIBLE + H_FRONT + H_SYNC + H_BACK;
 61
 62      localparam int V_VISIBLE = 480;
 63      localparam int V_FRONT   = 10;
 64      localparam int V_SYNC    = 2;
 65      localparam int V_BACK    = 33;
 66      localparam int V_TOTAL   = V_VISIBLE + V_FRONT + V_SYNC + V_BACK;
 67
 68      localparam logic [17:0] ADDR_CONTROL     = 18'h00000;
 69      localparam logic [17:0] ADDR_SCREEN_TYPE = 18'h00001;
 70      localparam logic [17:0] ADDR_ACTIVE_BUF  = 18'h00002;
 71      localparam logic [17:0] ADDR_TEXT_BASE   = 18'h02000;
 72      localparam logic [17:0] ADDR_IMG_A_BASE  = 18'h10000;
 73      localparam logic [17:0] ADDR_IMG_B_BASE  = 18'h20000;
 74
 75      localparam int TEXT_SIZE = 8192;
 76      localparam int IMG_SIZE  = 65536;
 77
 78      logic [10:0] h_count;
 79      logic [9:0]  v_count;
 80      logic [9:0]  x640;
 81      logic [9:0]  y480;
 82      logic visible_now;
 83      logic hs_now;
 84      logic vs_now;
 85
 86      logic display_enable;
 87      logic screen_type;
 88      logic active_buf;
 89
 90      logic text_we;
 91      logic img_a_we;
 92      logic img_b_we;
 93      logic [12:0] text_waddr;
 94      logic [15:0] img_waddr;
 95
 96      logic [6:0] char_col_now;
 97      logic [5:0] char_row_now;
 98      logic [2:0] glyph_x_now;
 99      logic [2:0] glyph_y_now;
100      logic [12:0] text_raddr_now;
101
102      logic in_img_now;
103      logic [7:0] img_x_now;
104      logic [7:0] img_y_now;
105      logic [15:0] img_raddr_now;
106
107      logic [7:0] text_char_s1;
108      logic [7:0] img_a_s1;
109      logic [7:0] img_b_s1;
110      logic [7:0] font_bits_s2;
111
112      logic visible_s1;
113      logic visible_s2;
114      logic in_img_s1;
115      logic in_img_s2;
116      logic screen_type_s1;
```

```
117      logic screen_type_s2;
118      logic active_buf_s1;
119      logic active_buf_s2;
120      logic display_enable_s1;
121      logic display_enable_s2;
122      logic [2:0] glyph_x_s1;
123      logic [2:0] glyph_x_s2;
124      logic [2:0] glyph_y_s1;
125      logic [7:0] text_char_s2;
126      logic [7:0] img_a_s2;
127      logic [7:0] img_b_s2;
128      logic hs_s1;
129      logic hs_s2;
130      logic vs_s1;
131      logic vs_s2;
132
133      logic [9:0] font_raddr;
134      logic glyph_on;
135      logic [7:0] img_pixel;
136      logic [7:0] final_pixel;
137
138      assign VGA_CLK = clk;
139      assign VGA_SYNC_n = 1'b1;
140
141      assign x640 = h_count[10:1];
142      assign y480 = v_count;
143
144      assign visible_now = (h_count < H_VISIBLE) && (v_count < V_VISIBLE);
145      assign hs_now = ~((h_count >= H_VISIBLE + H_FRONT) &&
146                        (h_count <  H_VISIBLE + H_FRONT + H_SYNC));
147      assign vs_now = ~((v_count >= V_VISIBLE + V_FRONT) &&
148                        (v_count <  V_VISIBLE + V_FRONT + V_SYNC));
149
150      assign text_we = chipselect && write &&
151                       (address >= ADDR_TEXT_BASE) &&
152                       (address < ADDR_TEXT_BASE + TEXT_SIZE);
153      assign img_a_we = chipselect && write &&
154                        (address >= ADDR_IMG_A_BASE) &&
155                        (address < ADDR_IMG_A_BASE + IMG_SIZE);
156      assign img_b_we = chipselect && write &&
157                        (address >= ADDR_IMG_B_BASE) &&
158                        (address < ADDR_IMG_B_BASE + IMG_SIZE);
159
160      assign text_waddr = address[12:0];
161      assign img_waddr  = address[15:0];
162
163      assign char_col_now = x640[9:3];
164      assign char_row_now = y480[8:3];
165      assign glyph_x_now = x640[2:0];
166      assign glyph_y_now = y480[2:0];
167      assign text_raddr_now = {char_row_now, 7'b0000000} + char_col_now;
168
169      logic [9:0] img_x_tmp;
170      logic [9:0] img_y_tmp;
171
172      always_comb begin
173          img_x_tmp = 10'd0;
174          img_y_tmp = 10'd0;
```

```
175          img_x_now = 8'd0;
176          img_y_now = 8'd0;
177          in_img_now = 1'b0;
178
179          if (screen_type) begin
180              // Fullscreen centered image
181              in_img_now = (x640 >= 10'd192) && (x640 < 10'd448) &&
182                           (y480 >= 10'd112) && (y480 < 10'd368);
183
184              if (in_img_now) begin
185                  img_x_tmp = x640 - 10'd192;
186                  img_y_tmp = y480 - 10'd112;
187                  img_x_now = img_x_tmp[7:0];
188                  img_y_now = img_y_tmp[7:0];
189              end
190
191          end else begin
192              // Gameplay image at right side
193              in_img_now = (x640 >= 10'd368) && (x640 < 10'd624) &&
194                           (y480 >= 10'd112) && (y480 < 10'd368);
195
196              if (in_img_now) begin
197                  img_x_tmp = x640 - 10'd368;
198                  img_y_tmp = y480 - 10'd112;
199                  img_x_now = img_x_tmp[7:0];
200                  img_y_now = img_y_tmp[7:0];
201              end
202          end
203
204          img_raddr_now = {img_y_now, 8'b00000000} + img_x_now;
205      end
206
207      byte_sdp_ram #(
208          .ADDR_WIDTH(13),
209          .DEPTH(TEXT_SIZE)
210      ) text_ram (
211          .clk(clk),
212          .we(text_we),
213          .waddr(text_waddr),
214          .wdata(writedata),
215          .raddr(text_raddr_now),
216          .rdata(text_char_s1)
217      );
218
219      byte_sdp_ram #(
220          .ADDR_WIDTH(16),
221          .DEPTH(IMG_SIZE)
222      ) img_a_ram (
223          .clk(clk),
224          .we(img_a_we),
225          .waddr(img_waddr),
226          .wdata(writedata),
227          .raddr(img_raddr_now),
228          .rdata(img_a_s1)
229      );
230
231      byte_sdp_ram #(
232          .ADDR_WIDTH(16),
```

```
233         .DEPTH(IMG_SIZE)
234     ) img_b_ram (
235         .clk(clk),
236         .we(img_b_we),
237         .waddr(img_waddr),
238         .wdata(writedata),
239         .raddr(img_raddr_now),
240         .rdata(img_b_s1)
241     );
242 
243     assign font_raddr = {text_char_s1[6:0], glyph_y_s1};
244 
245     font_sdp_rom font_ram (
246         .clk(clk),
247         .raddr(font_raddr),
248         .rdata(font_bits_s2)
249     );
250 
251     always_ff @(posedge clk) begin
252         if (reset) begin
253             h_count <= 11'd0;
254             v_count <= 10'd0;
255         end else begin
256             if (h_count == H_TOTAL - 1) begin
257                 h_count <= 11'd0;
258                 if (v_count == V_TOTAL - 1) begin
259                     v_count <= 10'd0;
260                 end else begin
261                     v_count <= v_count + 10'd1;
262                 end
263             end else begin
264                 h_count <= h_count + 11'd1;
265             end
266         end
267     end
268 
269     always_ff @(posedge clk) begin
270         if (reset) begin
271             display_enable <= 1'b1;
272             screen_type <= 1'b0;
273             active_buf <= 1'b0;
274         end else if (chipselect && write) begin
275             if (address == ADDR_CONTROL) begin
276                 display_enable <= writedata[0];
277             end else if (address == ADDR_SCREEN_TYPE) begin
278                 screen_type <= writedata[0];
279             end else if (address == ADDR_ACTIVE_BUF) begin
280                 active_buf <= writedata[0];
281             end
282         end
283     end
284 
285     always_ff @(posedge clk) begin
286         if (reset) begin
287             visible_s1 <= 1'b0;
288             visible_s2 <= 1'b0;
289             in_img_s1 <= 1'b0;
290             in_img_s2 <= 1'b0;
```

```
291             screen_type_s1 <= 1'b0;
292             screen_type_s2 <= 1'b0;
293             active_buf_s1 <= 1'b0;
294             active_buf_s2 <= 1'b0;
295             display_enable_s1 <= 1'b1;
296             display_enable_s2 <= 1'b1;
297             glyph_x_s1 <= 3'd0;
298             glyph_x_s2 <= 3'd0;
299             glyph_y_s1 <= 3'd0;
300             text_char_s2 <= 8'h20;
301             img_a_s2 <= 8'd0;
302             img_b_s2 <= 8'd0;
303             hs_s1 <= 1'b1;
304             hs_s2 <= 1'b1;
305             vs_s1 <= 1'b1;
306             vs_s2 <= 1'b1;
307         end else begin
308             visible_s1 <= visible_now;
309             visible_s2 <= visible_s1;
310             in_img_s1 <= in_img_now;
311             in_img_s2 <= in_img_s1;
312             screen_type_s1 <= screen_type;
313             screen_type_s2 <= screen_type_s1;
314             active_buf_s1 <= active_buf;
315             active_buf_s2 <= active_buf_s1;
316             display_enable_s1 <= display_enable;
317             display_enable_s2 <= display_enable_s1;
318             glyph_x_s1 <= glyph_x_now;
319             glyph_x_s2 <= glyph_x_s1;
320             glyph_y_s1 <= glyph_y_now;
321             text_char_s2 <= text_char_s1;
322             img_a_s2 <= img_a_s1;
323             img_b_s2 <= img_b_s1;
324             hs_s1 <= hs_now;
325             hs_s2 <= hs_s1;
326             vs_s1 <= vs_now;
327             vs_s2 <= vs_s1;
328         end
329     end
330 
331     always_comb begin
332         glyph_on = font_bits_s2[3'd7 - glyph_x_s2];
333         img_pixel = active_buf_s2 ? img_b_s2 : img_a_s2;
334 
335         if (!display_enable_s2 || !visible_s2) begin
336             final_pixel = 8'h00;
337         end else if (screen_type_s2) begin
338             final_pixel = in_img_s2 ? img_pixel : 8'h00;
339         end else if ((text_char_s2 != 8'h20) && glyph_on) begin
340             final_pixel = 8'hff;
341         end else if ((text_char_s2 != 8'h20) && !glyph_on) begin
342             final_pixel = 8'h00;
343         end else if (in_img_s2) begin
344             final_pixel = img_pixel;
345         end else begin
346             final_pixel = 8'h00;
347         end
348     end
```

```
349
350      always_ff @(posedge clk) begin
351          if (reset) begin
352              VGA_HS <= 1'b1;
353              VGA_VS <= 1'b1;
354              VGA_BLANK_n <= 1'b0;
355              VGA_R <= 8'd0;
356              VGA_G <= 8'd0;
357              VGA_B <= 8'd0;
358          end else begin
359              VGA_HS <= hs_s2;
360              VGA_VS <= vs_s2;
361              VGA_BLANK_n <= visible_s2;
362              VGA_R <= {final_pixel[7:5], final_pixel[7:5], final_pixel[7:6]};
363              VGA_G <= {final_pixel[4:2], final_pixel[4:2], final_pixel[4:3]};
364              VGA_B <= {final_pixel[1:0], final_pixel[1:0], final_pixel[1:0],
     final_pixel[1:0]};
365          end
366      end
367  endmodule
```