Word Game Engine: Wordle + Word Hunt

Team Members: Ben Li, Alice Hu

1. Introduction

We built a word game engine on our five-stage pipelined processor. We implement two popular word games: Wordle and Word Hunt. The user chooses a game via the main menu and interacts with the game using a keyboard, while outputs are shown on a VGA screen. All game logic is implemented in assembly instructions, which run on our processor.

2. Overall Project Design and Specifications

The user first selects a game from the main menu using the keyboard: pressing 1 directs them to Wordle, and pressing 2 directs them to Word Hunt. The respective game is immediately displayed on the screen.

2.1. Wordle

Wordle involves guessing a hidden five-letter word in six tries. A 6x5 grid is displayed to the user, in which the user types their guesses on the keyboard. When a user starts a new game, a random five-letter word is selected from a list of 500 words pre-loaded into RAM. For each five-letter word inputted, the word is automatically "submitted" upon reaching the end of the row, and the game displays color-coded feedback:

- Green (exact match): correct letter and correct position
- Yellow (partial match): the letter is in the hidden word but in the wrong position
- Gray (no match): the letter is not in the hidden word

Backspace is also supported to delete letters before submitting a word. If the user correctly guesses the hidden word within six attempts, the game ends.

If the user fails to guess the word after all attempts, the answer is displayed at the bottom of the screen, and the game ends. Now, pressing 0 returns to the menu, 1 resets Wordle with a new hidden word, and 2 jumps to Word Hunt.

2.2. Word Hunt

Word Hunt involves finding as many valid, connected words as possible in a 4x4 grid of letters. When a user starts a new Word Hunt game, a 4x4 board is displayed from a list of pre-made boards in RAM. The user has a cursor and navigates the grid using keyboard controls:

- WASD keys to move the cursor (up, down, left, right)
- Spacebar to select letters
- Enter to submit a word once a valid sequence is formed

On each letter selection, the game checks that the letter is adjacent to the last selected letter. When submitting a word, the game checks that the word is not a repeat submission and matches a word in the preloaded answer list. If valid, the word is accepted and the score is increased by the length of the word. As before, the user can navigate using the keyboard: 0 (menu), 1 (Wordle), and 2 (new Word Hunt game).

2.3. Memory Layout

Address	**Description**
0 - 999	Preloaded Wordle words (two entries per word), encoded to .mem using Python script
1000	MMIO region: keyboard scan code
1001 - 1030	MMIO region: Wordle letter inputs
1031 - 1060	MMIO region: Wordle color codes
1070 - 1074	Wordle answer letters

1100 - 1355	Word Hunt boards (64 entries per board, each board has 16 entries for the grid and 48 entries storing 24 valid answers)
1500 - 1515	MMIO region: 4x4 Word Hunt grid
1516 - 1563	48 entries storing 24 words in the grid
1564 - 1571	MMIO region: Word Hunt letter inputs
1572	MMIO region: Word Hunt selected cells
1573	MMIO region: Word Hunt last row selected
1574	MMIO region: Word Hunt last col selected

3. Inputs and Outputs

3.1. Inputs

All inputs come from a keyboard using the PS/2 protocol. We use the PS/2 interface provided in the lab, and also modified the data RAM so that on every clock cycle the last scan code from the keyboard is written to address 1000 in memory. Then, we use the *lw* instruction in assembly to read the scan code from RAM.

3.2. Outputs

Our primary output is a VGA display. We built on top of the VGA lab and used MMIO to map the values at specific memory addresses to sprites displayed at particular locations on the screen. In addition, we use register-mapped I/O in Word Hunt to map registers to content on the screen.

For Wordle, we have an array of letter scan codes of length 30 in RAM for the user's six guesses, as well as an array of color codes of length 30 for the color of each grid cell. The VGA controller maps the current *x* and *y* positions on the screen to a particular index in the letter array and color array, which gives a scan code that is then mapped to the corresponding sprite. The current *x* and

y positions are also used to compute the pixel within the 50x50 sprite. We similarly index into the color array to display the correct background color. These two MMIO arrays of letters and colors are populated by *sw* instructions in assembly after the user input is read from RAM via *lw*.

For Word Hunt, we use an array of letter scan codes of length 16 in RAM for the 4x4 grid, and an array of length 8 for the user's current word. The VGA controller uses the *x* and *y* positions on the screen to obtain the array index, map to the corresponding sprite, and compute the current pixel within the sprite. In addition, the lower 16 bits of one 32-bit word in RAM indicate which of the 16 grid cells has been selected. The VGA controller reads these 16 bits and sets a different background color for selected cells. Finally, we expose registers *$3* (cursor row), *$4* (cursor column), and *$22* (score) to the VGA controller to highlight the cursor cell and display the score.

4. Modifications to the Processor

4.1. Instruction Set

We modified the instruction set by changing the *sra* (shift right arithmetic) instruction to *srl* (shift right logical). For our answer lists, we store English words in a compressed format, with each 32-bit word in RAM containing four one-byte characters. To extract individual characters, we use a bitmask on the 32 bits, followed by a logical right shift. This required *sra* instead of *srl*.

4.2. Register File

We used an LFSR (Linear Feedback Shift Register) to generate pseudo-random numbers in our register file. This allows us to randomly select an index from our word list in Wordle, and randomly select a pre-made board in Word Hunt. We implemented the LFSR by XORing bits from a register to get the next pseudo-random value. Registers *$28* and *$29* were dedicated random registers that updated every clock cycle.

4.3. Memory-Mapped I/O

We incorporated keyboard input as a memory-mapped I/O device. We permanently tied address 1000 of the data RAM to the current keyboard input, such that every clock cycle, the value at address 1000 is updated with the latest PS/2 scan code.

We added four read ports to the data RAM to allow the VGA controller to read from multiple memory addresses. Wordle-related data, including guessed letters and colors, were stored in MMIO addresses. Word Hunt data, such as the current grid and selected letters, were also stored in memory-mapped locations. By increasing the number of read ports, the VGA controller could simultaneously fetch user inputs, Wordle color codes, and the Word Hunt grid from RAM.

5. Circuit Diagrams

We used the VGA display and PS/2 keyboard for I/O, and did not have other external circuits.

6. Challenges and Solutions

6.1. Word Encoding in Memory

For the word lists, we compressed English words into 32-bit data words, with each letter encoded as a byte, allowing us to store up to four letters per 32-bits. This format saves memory.

However, we accept user input one letter at a time. It made sense to store one letter per memory address, as this would be easier for MMIO when mapping between screen positions, memory addresses, and sprites. Comparing user inputs and compressed answers was a challenge.

We solved this problem by using bitmasks followed by a logical right shift to extract individual bytes/letters from compressed 32-bit data words. For example, 0x1C000000 can be ANDed with bitmask 0xFF000000 and shifted right to get 0x1C, the scan code value for the letter “A”. This also required modifying the instruction set to change *sra* to *srl*.

6.2. Word Comparison and Adjacency Logic

Both games involve complex logic, including exact and partial matching in Wordle, as well as adjacency checking in Word Hunt. We were able to successfully implement the backend game logic by organizing our code in a structured way and keeping detailed documentation with descriptive labels in assembly.

6.3. I/O

When reading keyboard input, we first noticed that one keypress caused multiple inputs to be registered due to the keyboard sending a character's downpress scan code, followed by a release code and the character's scan code again. We resolved this problem by repeatedly loading from MMIO address 1000 and waiting for a release code, then waiting for the input to change to an alphabetical scan code (or space/enter) before processing the input.

7. Testing Plan

7.1. Autotester

In initial stages, we used the autotester provided in the processor project to test simple assembly code, such as selecting a random word from memory in Wordle. We were able to verify register values, but this did not allow us to test interactive features.

7.2. LED Tests

Throughout the development and debugging process, we mapped bits from different registers to LEDs on the FPGA board. This enabled us to verify register values. We also used *$20* as a "checkpoint" register. By setting *$20* to specific values at different points in our assembly code, we were able to determine whether certain portions of code were reached.

7.3. Functional Tests

We also did extensive manual testing. For Wordle, the color codes were good indicators of whether our game logic was correct. For testing purposes, we also revealed the answer at the start of the game to verify the color codes. For Word Hunt, the score and highlighted cells were good indicators for the correctness of our adjacency checking logic and answer lookup logic.

8. Assembly Code Overview

All of our assembly code is in *main_game.s*.

8.1. Processing Keyboard Input

The input processing assembly logic is shared across the menu, Wordle, and Word Hunt. A loop repeatedly executes *lw* from address 1000 into *$5*, which contains the latest PS/2 scan code from the keyboard. The loop waits until the scan code is a release code. We then enter a secondary loop that continues loading the scan code, and waits until the scan code changes. The new scan code is checked for whether it is alphabetical or space/enter. If so, we jump to the appropriate function, and otherwise continue waiting for a release code.

8.2. Game Selection Menu

Our program starts at the menu, where we initialize two registers: *$25 = 0* to track the current game (0 = menu, 1 = Wordle, 2 = Word Hunt), and *$24 = 0* for whether to display the Wordle answer. The program waits until the user presses 1 (jump to Wordle) or 2 (jump to Word Hunt).

8.3. Wordle

Setup: At the start of a Wordle game, *$29* (which contains a random number every clock cycle) is used as an index into our word list, and we load a random five-letter word into *$2* and *$3*. Note

that, in our answer list, each 32-bit word encodes up to four English letters. MMIO addresses 1001 to 1061, storing user letters and color codes, are reset to zero. Register *$1* tracks the round.

State Tracking and Input Processing: For each round (i.e., five-letter word), we use registers to track the current letter position and a pointer (*$7*) to the user letter array (addresses 1001 to 1030). We enter a loop that waits for keyboard input, then load the input from memory (address 1000) and check if it is alphabetical. If so, we store the scan code in the user letter array at the address in *$7* and increment the letter position.

Word Comparison: Once five letters are inputted, we jump to the comparison function. Individual letters of the answer are extracted into five registers using bitmasks and shifts, and each letter in the user word is compared to each letter of the answer. The user letter is first compared to the answer letter in the same position, with a match resulting in a green color code. If they do not match, the user letter is compared with all other letters in the answer, with a match resulting in a yellow color code. Then, we store the color code (1 = green, 2 = yellow, 3 = gray) to the MMIO color code array (addresses 1031 to 1060).

We also use a register to count the number of "green" letters after every round. If all five user letters are exact matches, we jump to the end of the game. If the guess is incorrect and six rounds (stored in *$1*) have passed, we jump to the end of the game, load the correct answer into MMIO addresses (1070 to 1074), and toggle *$24* to 1 to tell the VGA controller to display the answer. Otherwise, we increment the round (*$1*) and jump to the start of a new round.

8.4. Word Hunt

Setup: We select a board from RAM by multiplying a random index in *$28* by 64 and adding it to address 1100. The grid and answer list (16 addresses for grid, 48 addresses for answer list) are

copied to a MMIO region (addresses 1500 to 1563) using a loop. Register *$18* contains a pointer to the user input list (addresses 1564 to 1571). Cursor position is tracked in *$3* (row) and *$4* (col). Register *$22* stores the score. MMIO addresses containing user input are reset to zero.

Input Processing: We enter a loop to wait for keyboard input. When W, A, S, or D is pressed, the cursor position (stored in registers *$3* and *$4*) is first checked to ensure it will not move out of bounds, and then *$3* or *$4* are modified based on the movement direction.

Letter Selection and Adjacency Checking: When space is pressed, the cursor position is converted to a single number (4 * row + column) and stored in *$8*, which is used to index into the list of grid letters (addresses 1500 to 1515), obtaining the letter at the cursor position. Then, a bitmask is generated in *$9* by shifting 1 by the index in *$8*. The bitmask is ANDed with the data at address 1572, which stores which cells have already been selected in the current user word. If the current letter has not been selected, we then load the last selected row and column from memory addresses 1573 and 1574, and take the differences between the rows and columns of the current and last selected cells. If either difference is greater than 1 or less than -1, the selection is rejected and we jump back to the loop waiting for keyboard input. Otherwise, the selection is valid: we store the selected cell's row and column at addresses 1573 and 1574, then OR the bitmask in *$9* with the data at address 1572 to mark the current cell as selected. Finally, we write the cell's letter (*$8*) to the user input list at the memory location pointed to by *$18*, increment *$18*, and jump back to the loop waiting for input.

Word Submission and Validation: Pressing enter loads the user input letters (addresses 1564 to 1571) in a loop and compresses the user input into two 32-bit values in *$20* and *$21* using OR and left shifts. In this loop, we also count the length of the user word in *$23*. Then, registers *$20*

and *$21* are compared to each English word (pair of 32-bit data words) in the answer list (addresses 1514 to 1563) in a loop. If a match is found, we increase the score stored in *$22* by the length of the word (*$23*). The program resets the user input letters (addresses 1564 to 1571) to zero and jumps back to waiting for input.

9. Improvements and Future Work

9.1. Expanded Word Database

We used a list of 500 Wordle words and a small number of Word Hunt boards. Adding support for an expanded word list, or loading/generating Word Hunt grids would improve the user experience of our game.

9.2. Word Hunt Timer and Difficulty Levels

A timer could be introduced to add time pressure to Word Hunt. Additionally, expanded Word Hunt boards such as 5x5 boards or 6x6 boards could be added to increase difficulty.

9.3. Expanded Selection of Word Games

Other word games, such as Anagrams, Crossword, Scrabble, or Hangman, could reuse core components such as the word dictionary, letter grid display, and keyboard input logic. These games could be implemented with minimal hardware changes.

10. Project Pictures