Project Report: MIPS Binary Game

Name: Hema Mekala

a) Program Description

This program is a single player "Binary Game" implemented entirely in MIPS assembly language, designed to run in the MARS environment. The game tests a player's ability to quickly convert numbers between their binary and decimal representations.

The program follows all project requirements, including:

- **Multiple Modules:** The program is split into multiple .asm files, each handling a specific piece of functionality (main.asm, convert.asm, drawBoard.asm, validateInput.asm, timer.asm, sound.asm, score.asm, generateProblem.asm).
- **Two Game Modes:** On launch, the user can select:
    1. **Binary to Decimal:** The game displays a random 8-bit binary number, and the user must enter the correct decimal equivalent (0-255).
    2. **Decimal to Binary:** The game displays a random decimal number, and the user must enter the correct 8-bit binary string.
- **Levels:** The game features 10 levels. The level number dictates the number of problems the user must solve to advance (Level 1 has 1 problem, Level 5 has 5 problems).
- **Randomization:** Each problem is randomly generated using the system time (syscall 30) as a seed for the random number generator (syscall 42), ensuring a different game every time.
- **Scoring & Feedback:** The user receives 10 points for each correct answer. Instant feedback ("Correct!" or "Wrong.") is provided after each problem.
- **ASCII Game Board:** A text-based board is drawn for each problem using ASCII characters
- to display the binary bits and decimal number.
- **Input Validation:** The program validates user input. In Decimal to Binary mode, it checks that the user has entered a valid 8-character string of '0's and '1's and prompts them to retry if the format is invalid.

Extra Credit Features Implemented

- **Graphic:** The ASCII board displays boxes for binary to decimal mode and lines for decimal to binary mode.

- **Sound:** The program uses the MARS MIDI sound capabilities to play a high-pitched "success" sound for correct answers and a low-pitched "fail" sound for wrong answers. This is handled by the sound.asm module.

- **Timer (Bonus Points):** The program uses the system clock to time how long a user takes on each problem. If the user answers correctly in under 10 seconds, they receive an additional 5 bonus points and a "FAST! +5 bonus!" message. This is handled by timer.asm and logic within main.asm.

b) Challenges

1. **Challenge: Function Calls and Stack Management**
    - **The Problem:** My first major bug was an infinite loop. The program would print "Line 1" and then get stuck. I discovered that when main.asm called jal gen_problem, the $ra (return address) register was set. However, gen_problem then immediately called jal gen_bits_random, which overwrote $ra. When gen_problem finished, it jumped back to itself instead of back to the main loop.
    - **The Solution:** I overcame this by implementing proper function prologues and epilogues. In any function that called another function (using jal), I first saved the $ra register to the stack (addi $sp, $sp, -4 and sw $ra, 0($sp)) and then restored it (lw $ra, 0($sp) and addi $sp, $sp, 4) before returning.
2. **Challenge: MIPS Calling Convention and Timer Logic**
    - **The Problem:** The timer-based bonus was extremely difficult to implement. My first attempt failed because the timer values were always zero. I discovered that syscall 30 (system time) returns the value in the $a0 register, not the $v0 (return) register.
    - **The Solution:** I fixed this by adding move $v0, $a0 to my timer_now_ms function, correctly passing the time back. My next bug was that the bonus always triggered. This was because my draw_binary_to_decimal function was saving and restoring the $s0 and $s1 registers. This erased the start/end time values I had just stored. I fixed this by simplifying the stack frame in those functions to only save $ra, allowing the main loop's $s0 and $s1 registers to be set and used as intended.
3. **Challenge: Blocking Syscalls and the "Hard Timeout"**
    - **The Problem:** My initial goal for the timer was a 15-second "hard timeout" that would interrupt the user. I realized this is nearly impossible with standard syscalls. syscall 8 (read_string) and syscall 5 (read_int) are blocking operations. They freeze the entire program, preventing any other code (like a timer check) from running.
    - **The Solution:** After realizing a "hard timeout" would require MMIO, I tried the "bonus points" implementation. This still fulfills the extra credit by using the timer, but it works with the limitations of blocking syscalls by simply checking the elapsed time after the user's input is received.

c) What I Learned

The key takeaways for me were:

- **MIPS Assembly:** I gained more proficiency in MIPS, including understanding the register file ($t, $a, $v, $s), directives (.data, .text, .globl), and common instructions (li, la, sw, lw, beq, jal, jr).
- **Stack and Memory:** I learned why the stack is critical. This project forced me to manage the stack manually for function calls, which solidified my understanding of how high-level languages handle function execution under the hood.
- **Modular Programming:** The requirement to use multiple files was a great lesson in software design. It forced me to think about shared data (using .globl on global variables) and create a "public" interface for each module (like gen_problem or validate_b2d).
- **Low-Level Debugging:** Debugging in assembly is a completely different skill. I learned to read assembler errors, trace program flow register by register, and use the "Messages" tab in MARS to find the exact line of failure.

d) Discussion of Algorithms and Techniques

The program's core logic resides in main.asm, which acts as the controller, and its "brain" is in the conversion and validation modules.

Program Flow

The program works as a set of nested loops:

1. **Main Loop (main_loop):** Asks the user for the game mode (1 or 2).
2. **Level Loop (level_start):** Runs from Level 1 to 10. It sets g_linesLeft equal to the current level.
3. **Problem Loop (line_loop):** Runs as long as g_linesLeft > 0. This is the main game logic:
    - It calls jal gen_problem to get a new random number.
    - It calls jal draw_ to show the board and get user input.
    - It calls jal validate_ to check the answer.
    - It provides feedback, sound, and scoring.
    - It decrements g_linesLeft and repeats.

How to Display the Board (drawBoard.asm)

The board is drawn using simple syscall 4 (print_string) calls.

- The .data section contains strings for the board parts: topLine: .asciiz "+---+---+...+\n" and pipe: .asciiz "| ".
- **ui_show_bits_board (Mode 1):** This function prints the topLine. Then, it enters a loop that runs 8 times. In each iteration, it prints a pipe and then calculates the address of the correct bit in g_currentBits (base address + loop index) to lb (load byte) and print it.

- **ui_show_decimal_board (Mode 2):** This function is simpler. It prints the topLine, a label with blank binary slots, a separator, and then "Decimal: ". It then uses syscall 1 (print_int) to print the value stored in g_currentDec.

Algorithms (convert.asm)

1. **bits_to_decimal(a0 = address of 8-bit string):** This function uses the iterative multiply-and-add algorithm.
    - result = 0
    - Loop 8 times:
        - result = result * 2 (using sll $t1, $t1, 1)
        - Load the next bit character
        - If bit == '1', result = result + 1 (using addi $t1, $t1, 1)
    - Return result in $v0
2. **decimal_to_binstr(a0 = decimal value, a1 = address of output buffer):** This function uses bitwise operations to convert a decimal number into an 8-bit string.
    - Loop i from 7 down to 0:
        - bit = (value >> i) & 1 (using srl and andi instructions)
        - If bit == 0, store '0' in the output buffer
        - If bit == 1, store '1' in the output buffer
        - Increment output buffer address
    - Store a null-terminator ('\0') at the end.

How to Validate Inputs (validateInput.asm)

Validation is the process of comparing the user's answer to the correct answer.

- **validate_b2d (Mode 1):**
    1. Calls bits_to_decimal using g_currentBits to calculate the correct decimal answer.
    2. Compares this result to the user's answer stored in g_userDec.
    3. Sets the global flag g_correct to 1 (if equal) or 0 (if not).
- **validate_d2b (Mode 2):**
    1. Calls binstr_to_decimal using the user's g_userBitsStr to convert their string into a number.
    2. Compares this number to the correct answer stored in g_currentDec.
    3. Sets g_correct to 1 or 0.