Razeen Rahman
CS2340.004
Prof. Alice Wang
17 October 2025

Binary Game

Description

The Binary Game program is a text based computer game written in MIPS Assembly that challenges users to convert decimal and binary numbers through different levels. The game's goal is to quiz and enhance the player's understanding of number systems by giving random questions. The program is developed entirely in a text based environment using system calls for input, output, and creating random numbers. When the application is run, users are presented with a menu that allows them to select Binary to Decimal or Decimal to Binary mode. With the Binary to Decimal mode, there is displayed a random 8-bit binary number, and the player must enter its equivalent decimal value. With the Decimal to Binary mode selected, a random decimal number between 0 and 255 is displayed, and the player must enter the equivalent 8-bit binary string. Each level contains the same number of questions as the current level, so the higher the player goes, the more questions are given per round. The game comes back to the main screen after completion of all ten levels for replay or to end the game.

The code was written in a modular structure broken into several assembly files for organizational and readability purposes. main.asm deals with the majority of the game's main logic including menu traversal, level progression, question loops, and user input handling. drawboard.asm deals with the text based graphics that are used to display the menu and level boards, whereas convert.asm deals with binary to decimal and decimal to binary functionality. io_utils.asm contains binary input validity logic to avoid users from giving anything else except valid 8-bit binary strings. Each module follows MIPS calling conventions.

Challenges

One of the biggest challenges I experienced in this project was implementing random number generation. Previous versions of the game produced random values erratically because the syscall used returned an unbounded integer that was sometimes greater than 255. Even after this was fixed, another issue appeared where the random numbers seemed to change during gameplay but repeated the exact same sequence every time the program was run. This was because the system calls in MIPS do not automatically reseed the random number generator between executions, so the same internal seed value was used at the start of the program each time. As a result, every run of the game generated the same sequence of “random” questions. When various system call codes were tested, the issue was resolved using SysRandIntRange and setting explicit lower and upper bounds for the range (0–256).

Synchronizing the ASCII board layout caused another major issue. Because MIPS shows characters linearly, text alignment like "Level: 6" or "Select Mode:" depended on the digits or spaces. This issue was resolved by establishing single and double digit level numbers that were separately pre and post spaced strings, which made it appear consistent regardless of the current level number.

A small but frustrating issue appeared towards the end of the project when the line "Question X of Level Y" began printing random huge question numbers. For example, "Question 8 of Level 2" or sometimes even numbers like “20329137.” The problem was in the question counter logic, where it calculated the question index using the remaining question count $s3. With $s3 decreasing every time, the calculation became offset and printed incorrect values. This was handled by introducing a special counter register $t8 that I incremented independently of $s3, so question numbers appear in the correct format and always in all modes. The second issue involved handling register safety between nested procedure calls. Functions sometimes overwrote return addresses or important registers, which resulted in crashes when trying to return from a syscall. To fix this, the program was changed to push $ra and all its corresponding $s registers onto the stack before each jump and link instruction so they could be preserved.

Implementing the input validation mechanism was another primary difficulty. Since user input is unpredictable, especially while entering binary strings, a special validation function was used to ensure the string consisted of eight characters exactly and that every character is either a '0' or '1'. At first, the main challenge was figuring out how to design the algorithm to properly loop through each character in the string and check its ASCII value without accidentally reading beyond the end of the input or missing the newline character. Initially, the function either skipped characters or failed to stop correctly at the null terminator. This caused the program to print inconsistent validation results. After several trials, I realized that using lb inside a loop with explicit checks for both the null and newline characters was the best way to traverse the user input safely without the issues I mentioned. This ensured that the loop could verify every binary digit accurately.

Reflection

Through the project, I gained a better idea of how high level logic is translated into low level machine code. I know how to handle stack frames, follow calling conventions, and structure modular assembly programs now because of this project. Creating the Binary Game also improved my reasoning ability for program flow when there are no easy high level control structures like loops or conditionals. Instead, I used branching and structured labels to mimic actual structured programming. I also became familiar with debugging within a low level system without the help of detailed debugging systems to identify logic bugs or syntax errors. Working with syscalls for random numbers, input, and output helped me understand how the computer actually performs basic tasks like printing or reading data behind the scenes for many programs.

Razeen Rahman
CS2340.004
Prof. Alice Wang
17 October 2025

Discussion

The Binary Game design revolves around nested control loops. The primary program loops between level advancement, main menu, and questions. The level and question cycles are determined by conditional branch comparisons of counters and jump instructions that switch between labeled sections of the code. The randomness in the questions is achieved through the use of random integers, where each random integer is either converted to binary or printed out to the console depending on the game's mode. The binary to decimal and decimal to binary conversion is completely within the convert.asm module, where bit manipulation and arithmetic logic is used to perform integer to binary and binary to integer conversion.

The convert.asm module works by simulating how binary representation is manually produced but using MIPS operations. To transform an integer into an 8-bit binary string (int_to_bin8), the code sets up a bitmask starting at 128 (which is 2^7). The code runs eight times, checking each bit of the integer through subtraction and comparison. In every iteration, the program verifies if the current position of the bits is adding to the number, so if the integer is large enough to hold that bit value, it writes ASCII '1' in the string and otherwise, it prints ASCII '0'. The mask is then divided by two by a right shift following each check, and iterations continue until eight bits are written. The outcome is a correctly formatted 8 character string which contains the binary representation of the original number. The bin8_to_int operation, being the inverse, is simply the reverse. It starts with an accumulator of zero, reads in each of the characters in the binary string, converts it from ASCII to its numeric value by subtracting 48, and lastly it shifts the accumulator one position left and adds the bit value. After all eight bits have been read in, the accumulator contains the decimal value of the binary input.

The io_utils.asm module also plays an important role as it provides user input verification and program security. The main routine, validate_bin8, is designed to assure that any binary string input by the user is exactly eight characters long and is made up of valid binary digits. The function operates on the string byte by byte using the lb (load byte) instruction. For every character, it checks the ASCII value with 48 ('0') or 49 ('1'). If it encounters any other character not in this range, the function directly returns 0 in $v0, meaning it was an invalid input. It also monitors the total number of allowed characters, ensuring that less than or greater than eight digits were not typed by the user prior to the newline terminator. If all eight characters are validated, then the function returns 1 and confirms the string is safe to use to continue the program. This check prevents the remainder of the game from processing faulty input that can lead to incorrect answers or unexpected behavior.

Together, io_utils.asm and convert.asm form the logical backbone of correctness for the game. The conversion routines are careful to correctly represent all numerical conversions between bases of numbers, while the check logic ensures that the user input is in the expected binary

Razeen Rahman
CS2340.004
Prof. Alice Wang
17 October 2025
format. The program within main.asm only exists to direct these two modules by passing arguments and getting their results.

For extra credit and to enhance the game visually, I added a text based graphical interface in drawboard.asm that uses ASCII characters to mimic borders, menus, and a title screen. The graphics were drawn using combinations of “+”, “-”, “|”, and “=” symbols that form boxes around the text to make the console display appear like a game menu. Spacing was calibrated by trial and error so that each border was perfectly aligned. The level, question numbers, and mode selection are dynamically inserted into the center of these boxes through formatted system calls from my main.asm file. This structure gave the game an easier look for the eyes even though it runs entirely in a text based environment.

Another thing I added for extra credit was sound feedback. A separate module called sound.asm produces distinct tones for correct and incorrect answers. This was achieved using the MARS system call for tones where a short frequency and duration are passed through registers to trigger beeps of different pitches. For instance, a higher frequency tone plays for a correct answer, while a lower-frequency one plays when the player answers incorrectly. These sounds are called from main.asm immediately after displaying the message for “Correct!” or “Not quite.” The addition of sound effects not only makes the game more engaging but also helps signal success or failure for users without even having to read the text.