The Design and Implementation of Blackjack Using Assembly Language

Mia Sideris
Ursinus College
CS 274
Collegeville, PA
misideris@ursinus.edu

Julian Faust
Ursinus College
CS 274
Collegeville, PA
jufaust2@ursinus.edu

Elijah Alexandre
Ursinus College
CS 274
Collegeville, PA
elalexandre@ursinsu.edu

***Abstract* — Using an 8086 Emulator and Turbo Assembly Language, we have designed and implemented a simple game of Blackjack. This game of Blackjack offers the user a choice among several game modes, including the computer betting mode, risk level and difficulty level. Additionally, the user can play with up to three decks at one time. Our software design decisions are flexible enough to allow for the implementation of multiple, unique game modes that present engaging, yet challenging experiences for the user.**

I. Introduction

The purpose of this project is to design and implement a simple Blackjack card game utilizing an 8086 Emulator and Turbo Assembly Language (TASM) to incrementally test each necessary procedure. Each aspect of the game was designed individually in six segmented parts, with the final measure being to combine each task to form the completed card game. As anticipated for the conventional game of blackjack, both active players aim to reach a combined card value that adds to 21 or the next closet possible value. Before beginning the game, the "human" player will input an initial betting amount between the range of $10 to the maximum of $100. Then, the user will have the option to choose between using 1-3 decks of cards for gameplay (note, no jokers are present). The computer opponent will also have the option to play one of three main modes of betting: conservative (under-bets human by 20%), normal: (matches human bet) and aggressive: (outmatches human by 30%). A computer risk level is determined by three numbers that must add to 100% for each category: keep hand, add card, and forfeit hand, that is implement following each turn. Lastly, the user is presented with a difficulty level option. That is, the user selects one of three modes which are in terms of the initial amount of money the computer is given. Easy mode is defined by the computer starting with half the human's amount of money, normal mode matches the human's amount, and hard mode starts the computer with 50% extra money than the human.

Each round begins with the human player indicating the amount they wish to bet from their available, accumulated amount of funds. Based on the computer's selected gameplay modes, the computer places a betting amount. Both players receive two cards (at random) to start their hands. Upon each turn, the player is presented with three options: keep their current hand, add card to attempt to reach 21, or forfeit their turn (assuming their current hand is generally close to reaching 21). If either player reaches 21 in their current hand, the turn is won for that player and the opponent's bet is taken and added to the winner's funds. However, if a player unknowingly exceeds 21 in their current hand, the player loses the turn, and funds are awarded to the opponent. In the event a player chooses to stand a turn, the current bet of each player remains untouched and will be evaluated following the next turn. Finally, the user is always presented upon the conclusion of each turn with the option to terminate the entire game, and not continue onto the next turn.

II. Software Design Decisions

The segmented portions were divided as follows: a) representing cards, bets and wins on screen, b) randomized card picking and tracking (single turn), c) tracking of bets and win (single turn), d) text user interface, e) computer betting behavior and finally, f) the fully integrated card game. To represent the mechanisms that will display cards, bets, and wins, first, a random number generation procedure is implemented (assuming a maximum of 16-bits). The random number generation procedure is designed to utilize Lehmer's algorithm to select one of the 52 cards contained within a standard deck. More precisely, the procedure is used to initially generate one of the four suites: hearts, diamonds, spades and clubs, and then called again to generate one of the ten possible numbers associated with each suit, in addition to kings, queens, jacks, and aces. Additionally, the implementation of a procedure called "erase_card" marks each previously used card as "0". This way, the program can easily identify cards that have been played against cards that have not been shown in gameplay yet. Thus, the active player is ensured each newly drawn card is regenerated randomly, being that one card cannot be used on multiple occasions within a full round.

Since the randomly generated number outputs in the form of a string, we must provide a mechanism to convert it to a numerical value that is able to be added and subtracted from with each newly drawn card value. This same mechanism will be applied to evaluate the amount of funds available to each user following the conclusion of each round. To address this concern, multiple conversion loops are implemented for each card value (ace-10), including king, queen, and jack. Additionally, the implementation of the procedure, "string_to_num," is used to compare the current string, say "A" for ace, to the corresponding ACSII value of 1 (or 11 depending on the player's current hand amount). The same process is executed for every available numerical card value each time a new card is drawn.

In order to identify if a valid bet amount is inputted by a user, a procedure named "ask_bet_amount," asks the user to input the amount of funds they wish to play with. Then, we implement various requirements for the input value. The first requirement is that the input value must be greater than two digits. If yes, the program will jump back to the start of a round. Next, the program checks if the bet amount is greater than four digits, also jumping to the start if it is. Finally, if the bet is found to be between the values 10-1000, then the program proceeds to begin the fundamental game loop. If the last number of the current bet amount is between 1000-1999 (greater than 1000), then the program jumps back to the start. The design of the game loop begins by asking the user for the bet amount. Note that the game modes and risk levels are determined prior to the start of each game. After the decided-on funds are inputted, the human player is always the one to initiate a round. The game loop essentially handles the logic of alternating between the human player's and the CPU's turn. The human player will begin their turn drawing two cards from the deck. Then, they will be presented with the question of whether they want to hit, stand or forfeit (resulting in immediate loss of round, and funds added to the CPU). If the player hits, the program will call the procedure that randomly generates a new card. If the player chooses to stand, their turn will be "removed." Following the player's strategic choice, the same process will be implemented onto the CPU.

III. Implementation

A significant advantage in the design of our Blackjack game is that it is heavily equipped to manage all sorts of invalid inputs and can address them properly before the user can begin the game. For example, if the user inputs an "unreasonable" bet amount (which we have defined as being larger than three digits in our program), then the user will be unable to begin the game until a valid input is recognized. This will prevent situations where a user may input an extremely large value, long enough to extend over multiple rows. This adjustment prioritizes the amount of available space in memory and prevents the user from causing the program to crash or freeze the terminal. Additionally, we have allocated two bytes of memory to both players' bet amount. Furthermore, the user will be unable to play if the input bet amount is less than $10. Therefore, the program accounts for situations where the user inputs nothing at all, a symbol, or some non-digit value, and a value that is arguably too small to consider betting with.

Though our program is robust enough to address many low-grade errors, one limitation is the length of execution time. Due to the number of loops present in our program, specifically to convert each number string into a value that is capable of arithmetic, the program operates fairly slow. Furthermore, before the program jumps to verifying that the string is converted, we also have multiple lines of code comparing the ASCII value to the numerical value. Only when these values are equal, can the program resume. This characteristic of our game may come at the expense of user engagement or enjoyment. Also, this aspect of our code takes up a significant number of lines that do not consider memory usage. As with most complex software design projects, this issue represents the famous decision of space versus scalability, or the number of problems addressed at a single time. Due to how many questions are asked of the user, we have chosen to adopt a design that highlights scalability over memory usage. It would be a greater risk to our program if we did not choose to adopt a scalable design style, given that the program cannot begin the game if the user input is invalid. In that case, the program would fail to function in the first place. Instead, given the complexity of this project, the amount of memory usage would turn out to be extensive either way.

One way to address the concern of slow execution time, or substantial memory usage is to create two separate procedures to resolve the issue, instead of several loops and lines to check and convert the card numbers. However, the difficulty that comes along with this implementation is that we must find some commonality between each numerical conversion so that it can be applicable to all numbers. Another issue that may arise here is that not only must the procedures be equipped to handle numerical conversion, but it must also adjust to letter conversions. For example, we must handle situations where "2" is initially a string, and we must convert it to the numerical ASCII value. Then, we also must consider instances where we are given a string "A" which must be converted to a value of either "1," or "11." We already have an existing procedure to handle when the user can choose when they want the ace to represent either option. Thus, these newly introduced procedures would likely have to call to that procedure, so that it recognizes which ASCII value to seek.

IV. Discussion & Conclusion

Through the process of designing and implementing a simple game of Blackjack, we have accomplished a multitude of achievements. First, though working in groups to tackle one complex task may mitigate the amount of work assigned to everyone, one issue typically overlooked is that at the end of the project. That is, we must all come together with our completed micro-tasks and implement them in a way that caters to everyone's choice of design. For instance, the implementation of the random number generator can be addressed in multiple ways. One of the first issues we experienced as a team was deciding on whether to a) randomly generate one of 52 cards, or b) randomly generate one suit, then one number. The second option would allow the program to appear more visually organized and create a smooth transition between random generation and string conversion, without the suit interfering. However, the first option limits the number of procedures, and addresses two problems into one concise way. Thus, we agreed our program was to adopt the first method to handle the generation of each card, and instead prioritize documentation to maintain organization.

Given that the development of this project entailed working alongside other programmers, arguably one of the most challenging aspects presented itself when we attempted to bring each portion of our code into one cohesive game. It is at this point that we had to address issues like the naming of each variable, and procedures. However, though resolving this minor issue took a considerable amount of time to tackle, it also forced each programmer to be precise and highly attentive to documentation. Therefore, in addition to accomplishing our individually assigned tasks, we also had to do it in a way that can be easily explained to someone that did not have to work on that portion. Furthermore, this issue incentivized us to work together as a team more than ever, so that everyone is informed on what procedures to call, and its according name. Thus, communication amongst our group came out to be a significant factor in the implementation of our Blackjack game.

Finally, to extend this project further, it would be interesting to implement a mechanism that would offer a heuristic rule that could improve the odds of either the computer, or opponent winning. Though the implementation of this mechanism would require the use of Artificial Intelligence, we could discuss various ways to accomplish this task. One technique could be that either player will have a higher likelihood of selecting a card best fit to their current hand. That is, if my current hand is ten, is there a way that the player can have an increased chance of choosing an ace out of the current deck. Additionally, this mechanism would also have to tie in cohesively with other features of the game, such as the current difficulty or risk level.