Blackjack in ASM

Pedro Gomez   
Ursinus College  
Computer ScienceCollegeville, PA  
pegomez@ursinus.edu

Timothy Jeon  
Ursinus College  
Computer ScienceCollegeville, PA  
tijeon@ursinus.edu

Abstract— In our assembly blackjack project, we prioritized structure and efficiency, ensuring clear code through naming conventions and detailed documentation. Challenges arose in designing the game loop with labels, yet we efficiently managed player and computer turns. However, issues with conditional jumps within functions and memory overflow errors posed significant challenges during implementation.

# Introduction (*Heading 1*)

# In this course project, our goal is to create a Blackjack card game through the 8086 emulator using assembly language where the user will play against a computer. We’ll first need to implement the user configuration which is where the user will need to choose the mode they’ll play in which includes initial human accumulated amount, number of card decks used, computer betting mode, computer risk level, and the difficulty. We’ll then need to implement the turn requirements where both the user and computer need the sufficient funds to continue, the turn is always started by the human, and the computer must bet all of its money despite the betting policy. We’ll then need to implement the turn actions which includes the user betting the amount they can, computer betting it’s amount based on the mode, both user and computer receiving two random cards, both user and computer having the three choices of keep current hand, add one more card to reach 21, and forfeit which means the computer wins, if hand value is over or lower than 21 for either user or computer, then that turn is won or lost for that player, if human decides to keep hand value, then bet is evaluated, and asking if the user wants to continue playing. Finally, we’ll need to implement the game end winning where the whole game ends if the user indicates they want to end the game, if there’s no more cards in the deck, or if the user or computer has no more money left. We’ll need to implement all of this to complete the Blackjack card game.

# To solve this, we’re going to first read and understand everything that’s needed for this game to work. This includes understanding all the user configurations, turn requirements, turn actions, and the game end and winning. Once we understand what we need to implement, we’ll then outline everything on the 8086 emulator by writing the methods needed to complete the program. Once we outline everything, we can implement the code and check if the program works by checking if the code either compiles or if the values stored are correct in memory.

# Software Design Decisions

For the deliverables, we both worked on different parts together or separately. For IPA 3, both Tim and Pedro worked on 3.1 and 3.2 which were representing cards, bets, and wins on screen and randomized card picking and tracking. For IPA 4, Tim worked on the 3.3 tracking of bets and wins while Pedro worked on the 3.4 text user interface. For IPA 5, Tim and Pedro both worked on 3.5 and 3.7 which were computer betting behavior and the integrated game which includes the completed game as well as the written report.

We decided to write our code with a mix of labels and functions in order to organize code and optimize the flow of the game. The primary role of labels in our code is to structure the flow of the game, while we use functions to abstract some of the data and memory manipulation going on behind the scenes. Upon some challenges midway through the project, we decided to use labels to jump to blocks of code conditionally. A strong learning point here was the benefits of using labels within the scope of a function, and the serious drawbacks of jumping to labels that are outside of a function.

# Implementation

One advantage of our design is the easy readability as anyone who views our code can understand what is going on due to the clear naming of labels and methods as well as the written documentation all over the code which explains what that specific part of the code is doing.

We faced a few challenges in our code. One challenge was deciding what the higher-level structure of the game loop should look like in a way that simplifies all of the work going on behind the scenes of the game. We settled on a structure that encompasses all of the main aspects of the game. The beginning of the round starts with a set of requirement checks to make sure both players have the means of playing, followed by labels which are used to differentiate between the player and computer turns. By using this label structure we can easily skip over unnecessary lines of code, such as skipping the computer's turn and going straight to the playerWin label whenever the computer forfeits. Using labels also made it much easier to jump between conditional blocks of code. We had originally started the assignment with a set of functions that were called from within the game loop labels, but this became an issue as functions do not support conditional jumps. As an initial solution, we created labels that other functions could jump to conditionally, but this resulted in some unwanted behavior. The conditional labels could not jump back into the function it was called from, causing some portions of the code in the function to not run.

Once all of the main game loop labels and input functions were completed, we were faced with another problem which posed a serious threat to the integrity of the program. Most variables used to store game information were two bytes in size, with respective  functions to ask the player to configure these variable values and store their inputs in memory. The flaw in this approach was assuming that the input “1” would be stored as its numerical value “0x01”. Instead, each character in a string takes up a byte in memory, so rather than us storing a numerical value, we store the hexadecimal ASCII value for each digit. This results in data overflowing into other portions of memory that end up breaking the game. To fix this, we created a buffer that would read the inputs, convert them into their appropriate hexadecimal values, and store the values in their appropriate addresses in memory.

# Discussions and Conclusions

First, we were able to implement the user configuration which includes the initial human accumulated amount between $10 and $1000, number of card decks between 1-3, computer betting mode of conservative, normal, or aggressive, computer risk level of keep current hand, add another card, or forfeit current hand, and difficulty of easy, normal, or hard.

Second, we were able to implement the turn requirements where the user always starts the turn, both the user and computer need to have the money to continue the turn, and the computer bets all its money despite the betting policy. After this, we implemented the turn actions which include user bet, computer bet based on mode, both players receiving two cards, and both players with options to either keep their current hand, add one more card, or forfeit, win, or lose if user or computer hits or exceeds 21, user keeping hand, and input to ask the user if they want to play next turn. Finally, we were able to implement the game end and winning where the whole game ends if the user indicates they want to quit where the winner is decided by who has the most wins, if there are no more cards in the deck, and if either the user or computer has no more money.

One lesson we learned from this project is that sometimes, going slow and steady is the correct approach to finishing code. Because assembly takes a while to run when going through the code, it can be frustrating at times to wait but this turned out to be very helpful as we can see when and where something goes wrong easily which helped us complete the blackjack game faster.

# Acknowledgment

Thank you to Timothy Jeon and Pedro Gomez for your contributions to the Blackjack card game. Both of you have demonstrated great work which is why this project was able to be completed.

# References

**These big brains of ours**