Description: There are two versions of the same number-guessing game. Both programs ask the user if they are ready to proceed. If they say yes, the program gives a set of 32 integers and asks if the user's chosen number exists in the set. If so, a value may be stored, depending on the version. If not, another set appears. After six iterations, the program displays the user's number, then asks if they would like to play again.

Challenges: The first challenge we faced was the logic. We noticed that there was a pattern in every set. Every number in the set has a particular bit that is always 1. The set consists of all possible combinations of that bit being 1, with all others being randomized. The user is essentially being asked if the bit in question is 1 or 0. This is how the number is mapped, and then it is displayed back to the user eventually.

The next challenge was calculating the numbers. We thought about doing so with an algebraic strategy. We attempted to figure out six formulas, one for each card. But then we realized all that could be synthesized into one master formula. The formula is the following:

a = n + 2^r + 2^r [floor((n)/(2^r))

a represents an element in a card. n represents the position of that element in that card. r represents the number of the card. For the first card, the formula will iterate from 0 to 31 with r set to 0. r will subsequently be set to 1 for the second card, 2 for the third, and so on through the 6th.

We also had to use an upper bound of 6 to randomize the cards, all the while making sure they did not repeat. We assigned a flag for every number in order to make sure the card would only be displayed once. The card flags also hold a value, which corresponds to the value of the card itself (1 for card 1, 2 for card 2, and so on). This means we were able to use the flags to calculate the result using integer arithmetic.

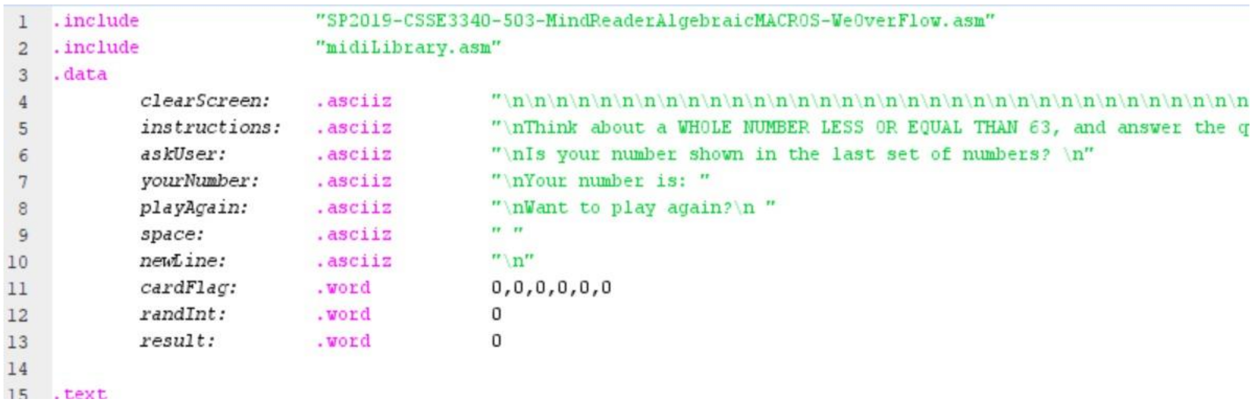
For the second version of the program, instead of using a formula, we used the randomized number as a shift amount, depending on the card.

The next challenge was the music. MARS is not able to handle MIDI playback well, but we did our best by using syscall 31. ADSLKFJADSKLFJAS

Learning experiences: I learned deeply about arithmetic and about the principles—specifically about the design principles of work, such as that simplicity favors regularity. It is crucial to keep a program simple and organized. I also learned how to make and implement macros, as well as how to use confirmation dialogs. We created seven macros that happened often in the game. It made the code easier to read and more organized. The most important ones were the card number evaluations and the master formula.

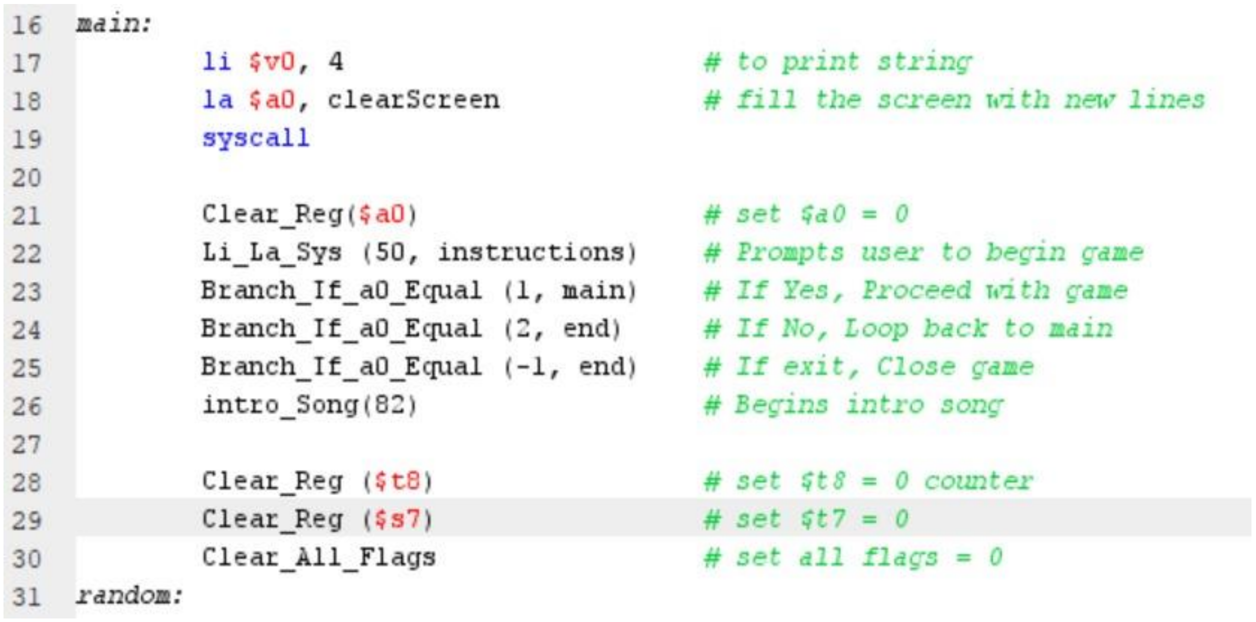
Algorithms and techniques:

Two include directives are in the program: one for the macros, another for the media library. All strings are accounted for.



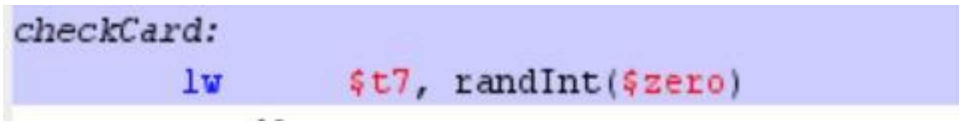
```
.include                "SP2019-CSSE3340-503-MindReaderAlgebraicMACROS-WeOverFlow.asm"
.include                "midiLibrary.asm"
.data
        clearScreen:    .asciiz         "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n
        instructions:   .asciiz         "\nThink about a WHOLE NUMBER LESS OR EQUAL THAN 63, and answer the q
        askUser:        .asciiz         "\nIs your number shown in the last set of numbers? \n"
        yourNumber:     .asciiz         "\nYour number is: "
        playAgain:      .asciiz         "\nWant to play again?\n "
        space:          .asciiz         " "
        newLine:        .asciiz         "\n"
        cardFlag:       .word           0,0,0,0,0,0
        randInt:        .word           0
        result:         .word           0

.text
```

Enough new lines are printed to make the screen clear. $a0 is cleared to clear the registry. The user is then prompted with a dialogue and song. All flags are cleared to 0. After the random number is created, it is stored in memory.



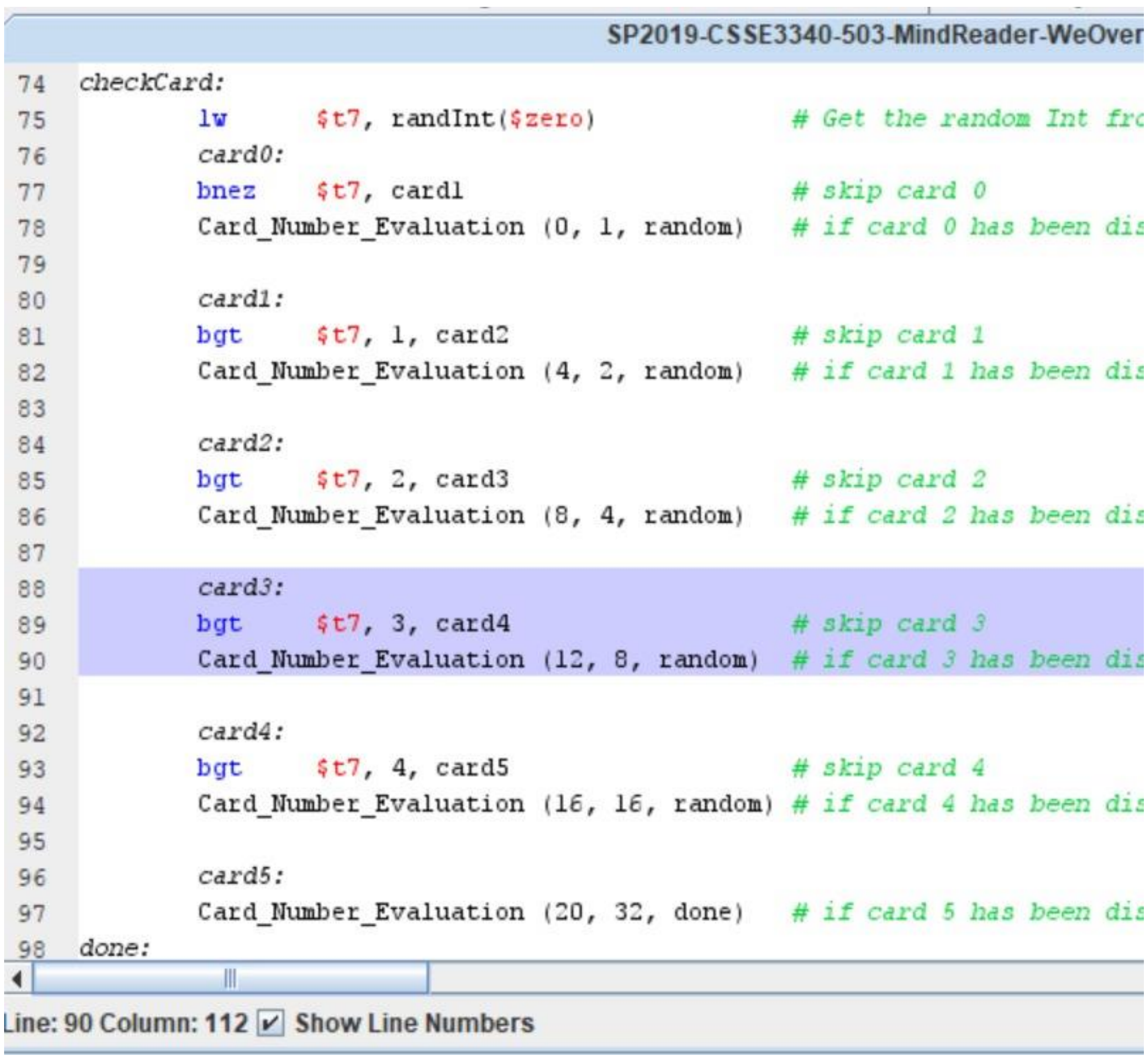
```
main:
        li $v0, 4                           # to print string
        la $a0, clearScreen                 # fill the screen with new lines
        syscall

        Clear_Reg($a0)                      # set $a0 = 0
        Li_La_Sys (50, instructions)        # Prompts user to begin game
        Branch_If_a0_Equal (1, main)        # If Yes, Proceed with game
        Branch_If_a0_Equal (2, end)         # If No, Loop back to main
        Branch_If_a0_Equal (-1, end)        # If exit, Close game
        intro_Song(82)                      # Begins intro song

        Clear_Reg ($t8)                     # set $t8 = 0 counter
        Clear_Reg ($s7)                     # set $t7 = 0
        Clear_All_Flags                     # set all flags = 0
random:
```

Then we jump and link to checkCard and log a random integer into $t7. We then load the random integer from memory into $t7.



```
checkCard:
        lw      $t7, randInt($zero)
```

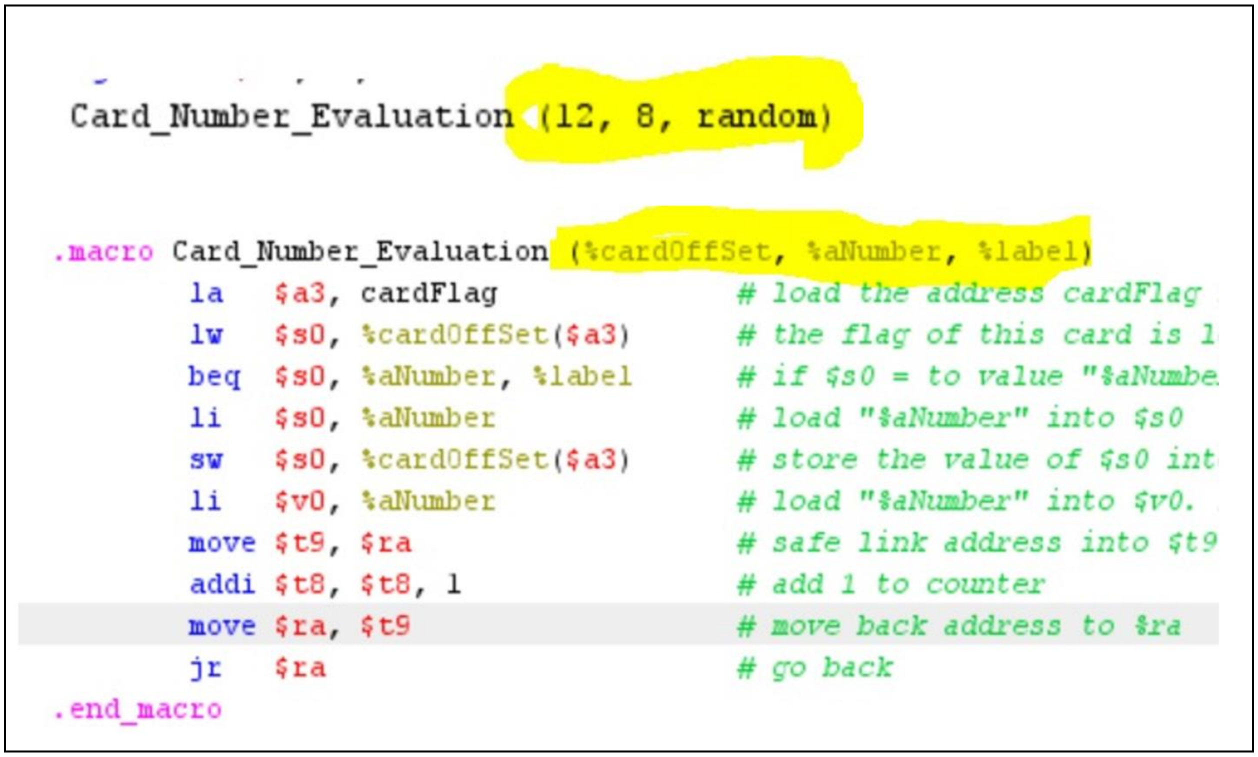
Let us say that $t7 has been loaded with 3 for the sake of demonstration. Since the value is not equal to 0, the card evaluation for the first card will be skipped, and the program will go to the next one. Then, since the value in $t7 is greater than 1, it will skip card 1 and go to card 2. Since the value is also greater than 2, the program will skip to card 3. Now, since 3 is not greater than 3, the branch will not occur here and the next statement will be executed.


SP2019-CSSE3340-503-MindReader-WeOver

```
checkCard:
        lw      $t7, randInt($zero)             # Get the random Int fro
        card0:
        bnez    $t7, card1                      # skip card 0
        Card_Number_Evaluation (0, 1, random)   # if card 0 has been dis

        card1:
        bgt     $t7, 1, card2                   # skip card 1
        Card_Number_Evaluation (4, 2, random)   # if card 1 has been dis

        card2:
        bgt     $t7, 2, card3                   # skip card 2
        Card_Number_Evaluation (8, 4, random)   # if card 2 has been dis

        card3:
        bgt     $t7, 3, card4                   # skip card 3
        Card_Number_Evaluation (12, 8, random)  # if card 3 has been dis

        card4:
        bgt     $t7, 4, card5                   # skip card 4
        Card_Number_Evaluation (16, 16, random) # if card 4 has been dis

        card5:
        Card_Number_Evaluation (20, 32, done)   # if card 5 has been dis
done:
```

Line: 90 Column: 112 ✓ Show Line Numbers

This statement is a macro that receives three values as arguments: a card offset, a number, and a label. The following code then gets executed.



```
Card_Number_Evaluation (12, 8, random)

.macro Card_Number_Evaluation (%cardOffSet, %aNumber, %label)
        la   $a3, cardFlag                # load the address cardFlag
        lw   $s0, %cardOffSet($a3)        # the flag of this card is 1
        beq  $s0, %aNumber, %label        # if $s0 = to value "$aNumbe
        li   $s0, %aNumber                # load "$aNumber" into $s0
        sw   $s0, %cardOffSet($a3)        # store the value of $s0 int
        li   $v0, %aNumber                # load "$aNumber" into $v0.
        move $t9, $ra                     # safe link address into $t9
        addi $t8, $t8, 1                  # add 1 to counter
        move $ra, $t9                     # move back address to $ra
        jr   $ra                          # go back
.end_macro
```

Load the address of the cardFlag. There are 5 values of cardFlag in memory, and we need to load the value for the card flag into $s0 with an offset. In this case, the value of the third card flag could be 0 or 8. If $s0 is equal to 8, branch to random. Else, continue with the next line of code. We load the value of 8 into $s0, and we store it into the card flag with its respective offset. Now we load immediate “aNumber,” which is 8, into $v0, because it could potentially be added to the result. Then we add 1 to $t8, which is the counter, and we branch back to $ra. Then $v0 gets moved to $s6 to hold the value of the register. Next, we jump and link to our element calculator, which is the function that prints the card according to the random number. Then we prompt the user if it appears in the last array. If yes, $t7 gets cleared, and the resulting value is loaded into $t7. $s6 is added into $t7 and $t7 is stored back into resulting memory. If $t8 is equal to 6, we branch to finalResult, and the number is printed on the screen. Then the user can conclude the game or continue playing.

Evaluations:

a. Alexander King – checked all code; created music and comments; audio/video. 10/10
b. McKenzie – helped to develop the code and debug it; documentation. 10/10
c. Suleman Itmer – did not contribute. 0/10
d. Hyeoncheol Kim – helped to develop the bit manipulation version of the code and condense it. 10/10