← Project 1: Twenty-one

Due Saturday, 10/5 by 11:59 PM (or late on Sunday)

In this project, you will be making a small textual game that is a simplified version of the card game Blackjack.

This project will have you focus on the following skills:

- The function calling convention we learned
- Input and output
- Translating high-level code (pseudocode) into low-level assembly
- Conditionals and loops
- Single-dimensional arrays

Description of the game

This is the overall description. For details on what your program should do specifically, see below.

Your program will play the role of the dealer.

1. Have an array of 52 cards.
2. Shuffle that array.
3. Deal two cards each to the player and dealer.
    - You only deal one card at a time. It should go player, dealer, player, dealer.
4. In a loop…
    1. Show both players' hands, including their total score.
    2. **Check the scores** (see below).
        - **It is possible for the game to end before the player takes a turn!**
    3. Let the player choose "hit" or "stand."
    4. If they "hit":
        - deal the player one card.
    5. Else, if the dealer's score is >= 17:
        - **see who's closer to 21** (see below) and **end the game.**
    6. Deal the dealer one card, if the dealer's score is < 17.

Checking the score

After showing the players' hands, you should check the score as follows:

1. If the player’s score is == 21:
    1. If the dealer’s score is == 21, **it’s a tie game.** The game ends.
    2. Else, **the player wins.**
2. Else, if the player’s score is > 21, **the player loses.**
3. Else, if the dealer’s score is > 21, **the player wins.**
4. Else, nothing happens and the game continues as normal.

The order of the conditions above is kind of important. When I implemented it I messed it up at first and it was possible to win in strange cases.

Seeing who’s closer to 21

In this case, both the player and dealer stand, and the winner is determined by who is closer to 21.

1. If the player’s score and dealer’s score are equal, **it’s a tie game.**
2. Else, if the player’s score is > than the dealer’s score, **the player wins.**
3. Else, **the player loses.**

Some example games

```
Dealer's hand: 10 4  = 14
Player's hand: 1 4  = 5
What would you like to do? (0 = stand, 1 = hit): 1
Dealer's hand: 10 4 6  = 20
Player's hand: 1 4 11  = 16
What would you like to do? (0 = stand, 1 = hit): 1
Dealer's hand: 10 4 6  = 20
Player's hand: 1 4 11 3  = 19
What would you like to do? (0 = stand, 1 = hit): 1
Dealer's hand: 10 4 6  = 20
Player's hand: 1 4 11 3 1  = 20
What would you like to do? (0 = stand, 1 = hit): 0
You tied!
```

```
Dealer's hand: 4 6  = 10
Player's hand: 4 6  = 10
What would you like to do? (0 = stand, 1 = hit): 1
Dealer's hand: 4 6 10  = 20
Player's hand: 4 6 5  = 15
What would you like to do? (0 = stand, 1 = hit): 1
```

```
Dealer's hand: 4 6 10  = 20
Player's hand: 4 6 5 9  = 24
You lost...
```

```
Dealer's hand: 1 4  = 5
Player's hand: 3 5  = 8
What would you like to do? (0 = stand, 1 = hit): 1
Dealer's hand: 1 4 2  = 7
Player's hand: 3 5 2  = 10
What would you like to do? (0 = stand, 1 = hit): 1
Dealer's hand: 1 4 2 10  = 17
Player's hand: 3 5 2 8  = 18
What would you like to do? (0 = stand, 1 = hit): 0
You won!
```

Implementation details

Approaching this code

Although this would be a very short program in a higher level language, it is many more lines in assembly. **Managing complexity** is a huge, huge part of writing in assembly.

DO NOT. TRY. TO WRITE. THE WHOLE PROGRAM. BEFORE TESTING IT. Test EVERY part AS YOU WRITE IT. Test your deck-shuffling function. Test your dealing-cards functions. *Test everything in isolation.* The easiest way to get overwhelmed and confused is by trying to write everything and finding out that nothing works.

`jal` should be your best friend. Start with `main` and work your way down. Do not put all the logic in main. Instead, start `jal`ing to functions which you have not yet written. Here's a sample `main` function. Feel free to use this.

```
.globl main
main:
        jal     shuffle_deck
        jal     deal_card_to_player
        jal     deal_card_to_dealer
        jal     deal_card_to_player
        jal     deal_card_to_dealer
```

```
_main_loop:
        jal     show_hands
        jal     check_scores
        jal     take_turns
        j       _main_loop
```

Keep splitting the program up like this. Eventually the functions you have to write will only be a few instructions long. That's good!

Use comments to keep track of which registers mean what. The register names do not convey meaning, so you have to make notes to yourself. But usually registers only hold those values for a short time anyway, so you'll have lots of comments.

The calling convention isn't just a set of rules; it makes it easier to write a program. When you follow the calling convention, questions like "which registers do I use? how do I know what value is in what register?" become very simple:

- **After a `jal`,** the `a, v, t` registers can contain anything.
  - So if you want a value to be in them, you have to put it in there yourself.
- **If you want to keep a value across a `jal`,** you must put it in an `s` register.
  - In that case, **you must push and pop that `s` register in the caller** along with `ra`.
- **You can always reuse registers.**
  - Honestly, I never use anything besides `t0, t1, t2`. Their values usually only last a couple lines.

Write pseudocode for yourself in comments BEFORE writing assembly. For example, when writing `shuffle_deck`, you might start it off as this, then after each line of comments, write the assembly that corresponds to it:

```
shuffle_deck:
        push    ra

        # for(i = 51; i >= 1; i--) {
        #     other_slot = random(0, i + 1);
        #     swap(deck[i], deck[other_slot]);
        # }

        pop     ra
```

```
    jr      ra # ALWAYS put this so you don't forget!
```

The deck and hands

You will need three arrays: one for the **deck** (the cards that have not yet been dealt), and **one for each player's hand.** You will also need to keep track of how many cards have been dealt from the deck and how many cards are in each player's hand.

The deck array should be initialized statically using `.byte` directives. It should have 52 cards. It will consist of **4 copies** of this:

```
    .byte 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 11
```

The player and dealer hands are mathematically unable to be longer than 9 cards, but give yourself some room and add a few more slots just in case something goes wrong. (it will.)

Shuffling the deck

You will use the Fisher-Yates shuffling algorithm to shuffle the deck. It's actually very simple, but translating it into assembly will take some trial and error. It works like so:

```
for(i = 51; i >= 1; i--) {
    other_slot = random(0, i + 1);
    swap(deck[i], deck[other_slot]);
}
```

- Note the condition here! We are not going down to slot 0.
- Since the deck is a byte array, address calculation is very simple. (B == 1.)
- Generating a random number can be done with **syscall 42.**
    - Pass the constant 0 for `a0`.
    - The upper bound `a1` is exclusive; if you call it with `a1 == 10`, you will get numbers in the range `[0, 9]`. That's why the code above uses `i + 1` as the upper bound.
- Swapping values in the array can be done easily in assembly without a temporary variable:
    - load the values from both slots into 2 registers;
    - then store those 2 register values in the opposite slots.

Drawing cards

When removing a card from the deck and placing it in a player's hand, you don't have to make it very complicated.

A simple solution is to keep track of the current position in the deck. That is, the first card that is drawn comes from `deck[0]`, then the next card drawn will be `deck[1]`, then `deck[2]` etc.

There is no need to "erase" the card from the deck. The "current position" in the deck is all the information you need to know; everything before the current position has already been drawn and copied into a player's hand.

Showing the players' hands

Only display as many cards are in the player's hand (i.e. if the player has only been dealt 3 cards, do not show 10). Show the individual cards in the hand, and then at the end of the line, show the total value of those cards:

```
Player's hand: 2 9 4 = 15
Dealer's hand: 10 3 1 = 14
```

Asking the player to hit or stand

You can simply use numbers to ask the player to hit or stand. You used the syscall to read numbers in lab 2. For example:

```
What would you like to do? (0 = stand, 1 = hit):
```

Ending the game

Display a message to tell the player the outcome of the game (won, lost, or tied).

Then use syscall 10 to exit your program and end the game.

Submission

You will submit a ZIP file named `username_proj1.zip` where `username` is your Pitt username. For example, I would name mine `jfb42_proj1.zip`.

See below if you have never made a ZIP file before.

Do not put a folder in the zip file, just the files you are submitting. When the grader opens the zip file, they should see the following:

- Your `twentyone.asm` file.
    - *Put your name and username at the top of the file in comments.*
- A `readme.txt` file. **DO NOT SUBMIT A README.DOCX. DO NOT SUBMIT A README.PDF. SUBMIT A PLAIN TEXT FILE. *PLEASE.*** It should contain:
    - Your name
    - Your Pitt username
    - Anything that **does not work**
    - Anything else you think might help the grader grade your project more easily

Submit here, just like you do with your labs.

Creating a ZIP file

1. In your file browser, select all the files you want to add to the ZIP file (the files listed above).
2. **Right click on the files,** and...
    - **Windows:** do **Send To > Compressed (zipped) folder**. Then you can name it.
    - **macOS:** do **Compress *n* items**. Then you can rename the `Archive.zip` file.
    - **Linux:** haha who knows

Grading Rubric

- **[16 points]:** Submission and style
    - **[6]:** You submitted your program properly (follow the directions above).
        - *This is all or nothing. Please, just follow the instructions.*
    - **[10]:** Your code is split into functions, and the functions use the calling convention we learned. **Don't keep global variables in registers!** Push/pop *s* registers and *ra* as needed!
- **[84 points]:** The game
    - **[24]:** Game flow
        - **[8]** No crashes encountered during normal operation
        - **[8]** Main loop works and ends when proper conditions are met
        - **[8]** When game ends, message is displayed, and syscall 10 is used to exit
    - **[24]:** Card management

    - **[12]** Deck is properly shuffled
    - **[8]** Arrays are used to hold players' hands
    - **[4]** Dealing a card correctly moves that card to a player's hand
  - **[24]:** Scoring
    - **[8]** Game tracks/calculates player scores (there are a couple solutions)
    - **[8]** Scores are compared properly after hands are shown
    - **[8]** "Closest to 21" is determined properly after both players stand
  - **[12]:** Input and output
    - **[4]** Prompts and accepts input for user to hit/stand
    - **[8]** Displays players' hands and scores