Project 1

<ARM BlackJack>

CSC 11 :48982

Amul Bham

Date: 11/8/2015

Title : ARM BlackJack

BlackJack is a simple card game in which the player competes against the dealer to get the higher hand but not over 21. Whoever goes over 21 loses. The player receives their first two cards and the dealer receives one. Based on the player's card total, they must decide to “hit” or “stay”, trying not to go over 21 but still attempting to have a higher hand than the dealer. The dealer is set up in a way where they never “hit” if they are at 17 or higher, therefore it is the player’s decision to hit or stay depending on if they think the dealer will bust or have a low card total.

For example: If the player has a hand of 14, they risk the chance of going over 21 on their next card draw. So they might decide to stay, despite having a low card total, in hopes that the dealer will ultimately bust. If they dealer has a 9 for their first card, they will keep drawing cards till they are at 17 or higher.

Basically the dealer is always playing risky, and it is up to the player to outsmart the dealer based on odds and knowing when to be aggressive or passive. Also note that if the player goes over 21, they lose no matter what because they draw their cards first. If the hand ends in a tie, no one wins. If a user has a blackjack (21), the dealer still has a chance of drawing a tie if they also have a 21.

This game provides a very similar experience to playing BlackJack in a casino and is intended to be a stimulating game based on odds and risk. The reason I choose BlackJack is because it is a relatively simple game to play, however requires a ton of logic to guide the program to the proper outcome. The game is challenging to program and I find joy translating the game to different programming languages. I must say, assembly by far proved to be the trickiest and provided me quite the challenge to get the logic and results correct. In addition, due to the complexity of the logic, the game ensures the use of all the programming concepts in a purposeful way. I often found myself stuck on a particular piece only to be relieved when I realized that we already learned how to fix the error in class, examples such as pushing the link register to the top of the stack for storage etc.

Summary:

Project Size : About 600 lines
Number of Variables : About 30
Number of Methods : 9
Repository:
https://github.com/amulbham/ASSEMBLY_CSC11/tree/master/proj/BlackJack_Proj_1

My project ended up being much larger than I originally anticipated coming in at about 600 lines of code between all the separate functions. This occurred simply because I under anticipated how difficult it would be to program all the different logic points in Assembly Language, something that proves to be simple in C++ can be exponentially more difficult in Assembly. For example, emulating a deck of cards proved to very difficult. At first, I tried to create an array and store random values between 2-53, and then have a player “draw” from this pool of random values to determine their card. However, I realized that I didn’t quite understand arrays fully to effectively implement this concept. So instead, I used a random number generator to give a value between 2-54 which was assigned to the player drawing. Even though their is bound to be duplicates, I figured it would suffice as Casino decks typically are made of four decks making duplicates a part of the game. Stimulating the concept of randomly drawing from a deck, as well as assigning this value a “face” and a card value proved to be the most difficult part of the game to program. This work can all be found within my get card function which took me around 6-8 hours to code and debug.

I also utilized open source code provided by Dr. Lehr to generate a true random number based on time. In addition, I used my previous C++ BlackJack project as a reference and “road map” for all the different logic points in the game. This proved to be crucial as it also allowed me to more easily visualize what needed to be done as C++ is far easier to interpret. Any time I found myself stuck or not knowing what to do, I would glance over the C++ code to give me some guidance. Although Dr Lehr constantly emphasizes the importance of coding assembly projects in C first, this project truly made me appreciate why as the project would have taken me far longer to code without C++ reference code.

I believe my game meets all the criteria for this project as I implemented all of the constructs we have learned thus far, and in a very innocent way. By innocent I mean, I wasn’t sitting their thinking about all the constructs I needed to use to get a good grade, I simply started coding and along the way I found truly meaningful ways to use different concepts and ideas we learned; in other words, nothing was forced. That is when I think true understanding is demonstrated, when you can utilize concepts learned in class without forcing it into your code, but naturally they find their way into the program. That's what I believe I did and why I believe my project meets the criteria for this assignment.

Sample Input/Output:

The executable file is called project1 -> type ./project1 in the working directory of the project to run the game

```
pi@raspberrypi ~/Desktop/ASSEMBLY_CSC11/proj/BlackJack_Proj_1/asm_files $ ./project1
Welcome to Amul Bham's BlackJack!
Would you like to hear the rules to play? 1 for yes, 2 for no
```

The game begins by asking the user if they would like to view the rules of BlackJack or jump straight into a game

```
Basically the point of the game is to get to 21
or as close as possible without going over
I will deal 2 cards from a regular deck of cards
depending on the total value of these cards, you must
decide if you want another card or stay where you are
Also, note that my job as a dealer is to always hit if my card
value is below 17, if i bust (go over 21) then you automatically win
```

The rules displayed

```
Now dealing your first two cards...
Queen of hearts
4 of cloves
card total: 14


Now dealing my first card...
Ace of diamonds
card total: 11


Would you like to hit(1) or stay (2)
```

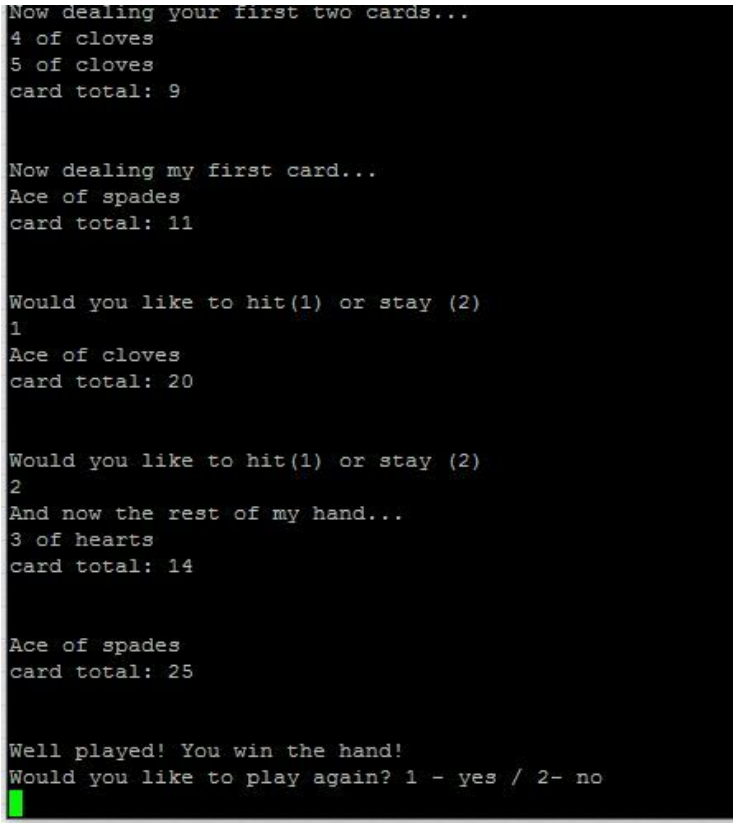
After the greeting and the rules, the first two cards of the player are outputted along with the first card of the dealer hand

The player must decide if they would like to hit(1) or stay(2) based on their current card value and the first card of the dealer.

```
Would you like to hit(1) or stay (2)
1
King of diamonds
card total: 24


You busted! Better luck next time!
Would you like to play again? 1 - yes / 2- no
```

If the player busts (goes over 21), then the program skips to the _lose function, the rest of the dealer hand is skipped, leaving the player to decide if they would like to play again

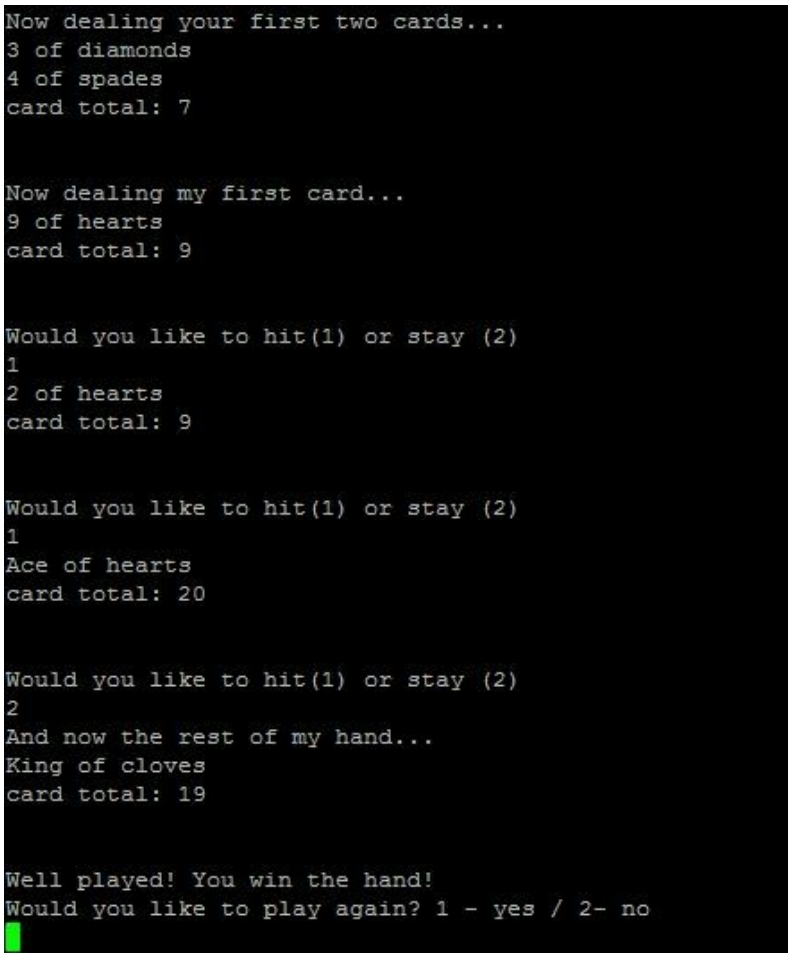


If the player decides to stay, or hit but not go over 21, then the rest of the dealer hand is displayed. The dealer will keep drawing until their card value is >= 17

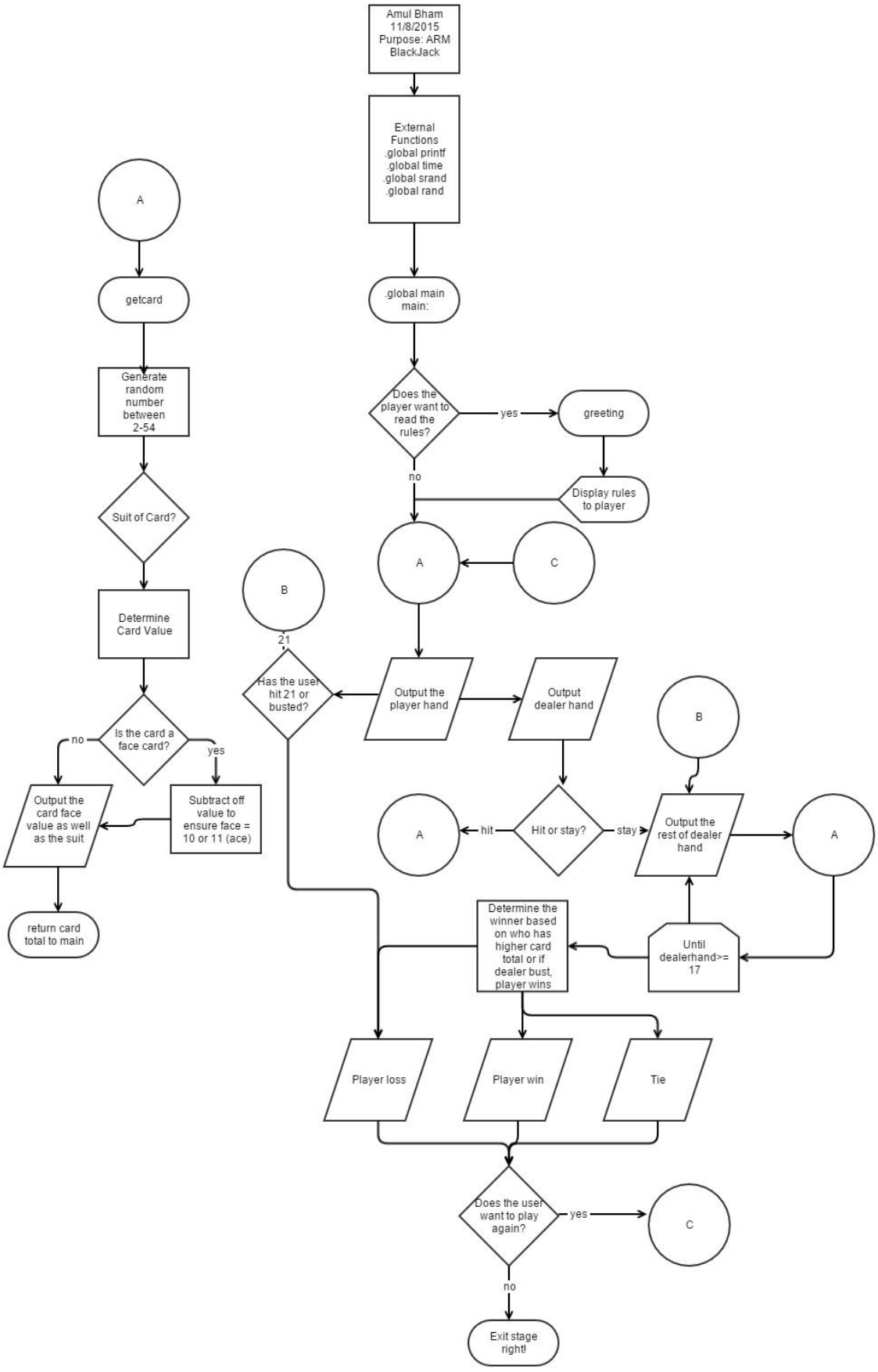
At this point, the winner is determined based on who had the higher card total but not over 21

In this example, the dealer busted so the player automatically wins

The player is prompted to play another game or exit the program



← Full Game


Amul Bham
11/8/2015
Purpose: ARM
BlackJack
External
Functions
.global printf
.global time
.global srand
.global rand
.global main
main:
Does the
player want to
read the
rules?
yes
greeting
Display rules
to player
no
A
C
B
21
Has the user
hit 21 or
busted?
Output the
player hand
Output
dealer hand
B
A
hit
Hit or stay?
stay
Output the
rest of dealer
hand
A
Until
dealerhand>=
17
Determine the
winner based
on who has
higher card
total or if
dealer bust,
player wins
Player loss
Player win
Tie
Does the user
want to play
again?
yes
C
no
Exit stage
right!
A
getcard
Generate
random
number
between
2-54
Suit of Card?
Determine
Card Value
Is the card a
face card?
no
yes
Output the
card face
value as well
as the suit
Subtract off
value to
ensure face =
10 or 11 (ace)
return card
total to main

Pseudocode:

First the user decides if they want to hear the rules of or not
if so, branch to the greeting function to display the rules
else, branch to the start of the game

The player then receives their first two cards
Cards are randomly selected using random number generator, assignment a suit, a card value and determines if the card is a face. Branch to he get card function

The dealer receives their first card
branch to the get card function

Player must now decide if they want to hit or stay
Program calculates the value of each player's hand after each card draw -> get card

if the user decides to hit, they will continue to receive cards till they are over 21, they decide to stay, or hit a BlackJack.

else, the dealer receives their next cards, and continues until they are at 17 or over.

If player or dealer bust, opposing player wins -> the calculations are ignored if either player busts and the game skips to the output

else, if both players have not gone over 21, a winner is determined.

Program calculates winner depending on who had the higher card total

if user has a higher card total, they win

else if the dealer has a higher card total, or the player has gone over 21, the dealer wins

User is prompted to play again or walk away

if user walks away -> exit the program

else, game loops to the start of the game loop, registers are reset for next hand

Major Variables:

Variable Name	Description	Location
pcard_total	stores the value of the players current card	.global main
dcard_total	stores the value of the dealer's current card	.global main
card_1:	keeps a running total of the player's hand	.global main
card_2:	keeps a running total of the dealer's hand	.global main
pagain	determines if the user would like to play another hand	.global main
address_of_message address_of_finish address_of_lose address_of_win address_of_tie address_of_plose21 address_of_plose21	displayed various messages regarding the outcome of the game, informs the user if they won, lost, tie, hit a blackjack etc.	.global main
address_of_return: address_of_opening: address_of_opening2: address_of_continue: address_of_continueA: address_of_hitorstay: address_of_firstcards: address_of_dfirstcards:	displayed various messages throughout the game, including the intro message, informing the user of what point the game currently is at, if they want to hit or stay, etc	.global main

output_hearts output_diamonds output_cloves output_spades	stored the string text of each of the four suits that the card could possibly be	.global getcard, .global dcard
output_jack output_queen output_king output_ace	stored the string text for each of the face cards in the case that the player or dealer draws a face	.global getcard, .global dcard
address_of_return	used to store the link register	.global main

Assembly Constructs Applied:

Concept	Implementation
Using link register	Stored the lr upon entering a function and then popped back the lr upon leaving the function Can be seen in my greeting function, I also utilized push and pop commands in my get card function
Passing registers as Parameters for a function	Before entering a function, I made sure beforehand that the proper variables were stored in R0-R3 to ensure they were passed to the get card and dcard functions appropriately for modification, can also be seen upon entering printf and scanf, I loaded the registers with the appropriate outputs, or format for scanf, prior to using them .
Branching : bge, bl, blt, beq, bx,b compare: cmp	Utilized branching like second nature all throughout my program, used it to make decisions, act like if else statements, switches, basically anywhere I needed the program to make a decision based on the input. I also used the bl command to branch to other functions while also passing the link register. Based on certain circumstances, I used branching to move fluidly and effectively around my program

str and ldr - store and load register	Used str command to preserve my link register upon entering a function. Used the ldr command all throughout my program to store memory addresses in certain registers for modification or passing to functions.
Looping using labels and branching	I used labels at the start of major points in the program to save writing excess code, for example the hit or miss part of the program can be looped until the player decides to stay or bust, instead of just rewriting the code, I used a label and a branch statement to continuously reuse this code given the user wants to keep hitting. I used this concept in many other parts of my program including the end when the user decides if they would like to play again, I simply created a label at the start of the game, and branch to that label if the user would like to play again.
Functions	Utilized functions to separate logic points in my program to streamline the code and make it more modular. For example, I kept the greeting function separated since it is essentially a wall of text explaining the rules, so instead of having that in my main function, where it takes up space and gets in the way of readability. Also calculating the card value and card total required a ton of code and logic that I kept separated from the main function, this allowed me to focus on getting the logic correct in that block of code instead of having it all jumbled up together in my main function. This allowed my main function to simply guide the program while my outside functions did the more complicated tasks. It also allowed me to more easily detect where a bug was arising from as I could individually debug functions.
External Functions	I used scanf and printf to read user input and output text to the screen all throughout my program. In addition, I also used the srand and time functions to bring in the current time and then plant the random number seed inside of the rand function which I used to generate a random number representing a card draw

Setting Flags using cmp	I used the cmp to compare numbers against register values in order to guide my program to the next step. For example, the dealer keeps drawing cards till their card total is at or above 17 so to ensure this, I used a cmp R1, #17 to set the flag, if the value was greater than or equal (bge) the program would go to the next step, else blt was executed and the program looped . Or when determining the winner of the game, I simply used cmp R1, R2, in other words I just directly compared the card values of the dealer and player, whoever was high was the winner
Addition, subtraction of values in registers	Used arm subtraction and addition all over my program to keep a counter, add the card values together, set loop conditions etc.

References:

1.) I utilized the DivMod function as well as parts of the random number generator generously provided by Dr. Lehr. Used to generate a random number to represent a random card drawn from a deck, the divmod function was used in tandem with the random number generator to obtain a range 2-53 using the modules command
   a.) https://github.com/ml1150258/LehrMark_CSC11_48982/blob/master/Class/Week9/randTest.s
   b.) https://github.com/ml1150258/LehrMark_CSC11_48982/blob/master/Class/Week9/mainDivModFuncV2.s
2.) I previously coded a game of blackjack in C++, I used this previous code as a template for the all the logic points in the game, as well as an overall reference.
   a.) https://github.com/amulbham/CSC-17A/tree/master/proj/BlackJack_Project_

Program:

Main:

```
/*Name: Amul Bham
Purpose: Develope BlackJack game in assembly language
Date: 11/5/2015
Main function of Blackjack Project*/

.data
/*address of return for link */
.balign 4
return: .word 0
/*Below are all the text I had to output to the user, or the format for input for scanf*/
.balign 4
opening: .asciz "Welcome to Amul Bham's BlackJack!\n"

.balign 4
opening2: .asciz "Would you like to hear the rules to play? 1 for yes, 2 for no\n"

.balign 4
continue: .asciz "%d"

.balign 4
firstcards: .asciz "Now dealing your first two cards...\n"

.balign 4
dfcards: .asciz "Now dealing my first card...\n"

.balign 4
hit: .asciz "Would you like to hit(1) or stay (2)\n"

.balign 4
message: .asciz "card total: %d\n\n\n"

.balign 4
finish: .asciz "And now the rest of my hand...\n"

.balign 4
win: .asciz "Well played! You win the hand! \n"

.balign 4
lose: .asciz "You lose! Better luck next time!\n"
```

```
.balign 4
tie: .asciz "Looks like we have a draw...\n"

.balign 4
lose21: .asciz "You busted! Better luck next time!\n"

.balign 4
playagain: .asciz "Would you like to play again? 1 - yes / 2- no\n"

/*Below are all the memory addresses used for user input*/
.balign 4
continueA: .word 0

.balign 4
hitorstay: .word 0

.balign 4
pagain1: .word 0

/*Card values - dealer card total and player card total*/
.balign 4
card1: .word 0

.balign 4
card2: .word 0

.balign 4
total: .word 0

.balign 4
total2: .word 0

.text

.global main

main:
/*store the link register*/
ldr r1, address_of_return
str lr, [r1]

/*Output the greeting, and rules if the user does not know how to play*/
ldr R0,address_of_opening
bl printf

ldr R0, address_of_opening2
bl printf
```

```
ldr R0, address_of_continue
ldr R1, address_of_continueA
bl scanf

ldr R6, address_of_continueA
ldr R6, [R6]
cmp R6, #2
BGE _gloop
bl _greeting

_gloop:
/*Variables to track the players running card totals, reset after each hand*/
ldr r8, card_1
ldr r8, [r8]

ldr r9, card_2
ldr r9, [r9]

mov r9, #0
mov r8, #0

_start:
/*Output the players first two cards*/
ldr R0, address_of_firstcards
bl printf


mov R0, #2
ldr r1, pcard_total
ldr r1, [r1]

bl _getcard

mov r8,r1
ldr r0, address_of_message
bl printf

/*If the user has hit a 21 on the first 2 cards, skip the hit or stay portion and finish dealers hand*/
cmp r8, #21
BEQ _dealerhand

/*Output the dealers first card*/
ldr R0, address_of_dfirstcards
bl printf

mov R0, #3
ldr r1, dcard_total
ldr r1, [r1]
```

```
bl _dcard
mov r9, r1
ldr r0, address_of_message
bl printf

/*After dealer first card, the user must decide to hit or stay, keeps looping till
user hits 21, busts, or decides to stay*/
_hit:
ldr r0, address_of_hit
bl printf

ldr r0, address_of_continue
ldr r1, address_of_hitorstay
bl scanf

ldr R7, address_of_hitorstay
ldr R7, [R7]
/*If the user decides to hit, finish the rest of the dealer hand*/
cmp R7, #1
BGT _dealerhand
mov R0, #1
ldr r1, pcard_total
ldr r1, [r1]
bl _getcard
/*Output the card total, store the card total in r8*/
add r1, r1, r8
mov r8, r1
ldr r0, address_of_message
bl printf

/*Determine if the user has busted or hit 21, else go back to hit or stay*/
cmp r8, #21
BEQ _dealerhand
BLT _hit
BGT _plose21

/*The dealer continues their hand, loop until their card total is 17 or higher*/
_dealerhand:
mov r5, #8
ldr r0,address_of_finish
bl printf
_newcard:
mov R0, r5
ldr r1, dcard_total
ldr r1, [r1]
bl _dcard
add r1, r1, r9
mov r9,r1
```

```
ldr r0, address_of_message
bl printf
add r5, r5,#4
cmp r9, #17
/*If card total is less than 17, loop back for another card*/
bge _end
blt _newcard

/*Output the results of the game to the user*/
_end:
/*If the dealer has busted, the user automatically wins*/
cmp r9, #21
bgt _pwin

/*Else, determine whose card value was higher*/
cmp r8, r9
bgt _pwin
beq _tie
blt _plose

/*Output for the results of the game, win lose or tie*/
_plose:
ldr r0, address_of_lose
bl printf
b _playagain

_pwin:
ldr r0, address_of_win
bl printf
b _playagain

_plose21:
ldr r0, address_of_plose21
bl printf
b _playagain

_tie:
ldr r0, address_of_tie
bl printf
b _playagain

/*After determining a winner, let the player decide if they would like to play again*/
_playagain:
ldr r0, address_of_playagain
bl printf

ldr r0, address_of_continue
ldr r1, pagain
```

```
bl scanf

ldr r4, pagain
ldr R4, [R4]

cmp r4, #1
/*Branch to the start of the game loop if yes, else exit the program*/
beq _gloop
b _exit


_exit:
ldr lr, address_of_return
ldr lr, [lr]
bx lr


/*Addresses for all my significant memory allocations*/
address_of_return: .word return
address_of_opening: .word opening
address_of_opening2: .word opening2
address_of_continue: .word continue
address_of_continueA: .word continueA
address_of_hitorstay: .word hitorstay
address_of_firstcards: .word firstcards
address_of_dfirstcards: .word dfcards
address_of_hit: .word hit
card_1: .word card1
card_2: .word card2
pcard_total: .word total
dcard_total: .word total2
pagain: .word pagain1
address_of_message: .word message
address_of_finish: .word finish
address_of_lose: .word lose
address_of_win: .word win
address_of_tie: .word tie
address_of_plose21: .word lose21
address_of_playagain: .word playagain




/*External Functions*/
.global printf
.global time
.global srand
.global rand
```

Greeting Function:

```
/*Greeting function - displays the rules to the user if they are not familiar*/
.data
.balign 4
return2: .word 0

.balign 4
message1: .asciz "Basically the point of the game is to get to 21\nor as close as possible without going over\n"

.balign 4
message2: .asciz "I will deal 2 cards from a regular deck of cards\ndepending on the total value of these cards, you must\n"

.balign 4
message3: .asciz "decide if you want another card or stay where you are\n"

.balign 4
message4: .asciz "Also, note that my job as a dealer is to always hit if my card\nvalue is below 17, if i bust (go over 21) then you automatically win\n\n\n\n"

.text
.global _greeting

_greeting:
/*Output the rules to the user*/
ldr r1, address_of_return2
str lr, [r1]

ldr R0, address_of_message1
bl printf

ldr R0, address_of_message2
bl printf

ldr R0, address_of_message3
bl printf

ldr R0, address_of_message4
bl printf
/*Link back to the main function*/
ldr lr, address_of_return2
ldr lr, [lr]
bx lr
```

```
address_of_return2: .word return2
address_of_message1: .word message1
address_of_message2: .word message2
address_of_message3: .word message3
address_of_message4: .word message4


/* External */
.global printf
.global scanf
```

Get Card Function:

```
/*Function used to generate a random card value, assign that card value a suit, and then add the card value to the players running card total*/
.data

.balign 4
current: .asciz "%d"
/*Output for each face value and suit value*/
.balign 4
hearts: .asciz " of hearts\n"
.balign 4
diamonds: .asciz " of diamonds\n"
.balign 4
cloves: .asciz " of cloves\n"
.balign 4
spades: .asciz " of spades\n"

.balign 4
jack: .asciz "Jack"

.balign 4
queen: .asciz "Queen"

.balign 4
king: .asciz "King"

.balign 4
ace: .asciz "Ace"


.text

.global _getcard
_getcard:
        /*push the lr to the top of the stack*/
```

```
push {lr}
	/*Setup a loop counter in r4, keep a running card total in r5*/
	mov r4,r0
	mov r5, r1

	mov r0,#0		/* Set time(0) */
bl time		/* Call time */
	bl srand		/* Call srand */

loop_rand:
	bl rand		/* Call rand */
	mov R1,r0,ASR #1		/* In case random return is negative */
	mov r2,#52

	bl divMod		/* Call divMod function to get remainder */
	/*Add two to the random number as a deck of cards starts at 2*/
	add R1,#2

	/*Based on the value 2-53, the card is assigned a suit*/
	cmp R1, #14
	BLE _hearts

	cmp R1, #27
	BLE _diamonds

	cmp R1, #40
	BLE _cloves

	BGT _spades
	/*Hold the suit in r3, subtract accordingly so the card value falls between 2-14*/
	_hearts:
	mov r3, #1
	b _continue

	/*After the suit is determined, a value must be applied if the card is a face card,
	else the value of the card is the number representation*/
	_continue:
	cmp R1,#14
	BEQ _sub4
	cmp R1,#13
	BEQ _sub3
	cmp R1,#12
	BEQ _sub2
	cmp R1,#11
	BEQ _sub1

	ldr R0, current_card
	b _addtotal
```

```
_diamonds:
mov r3, #2
sub R1,#13
b _continue


_cloves:
mov r3, #3
sub R1,#26
b _continue

_spades:
mov r3, #4
sub R1,#39
b _continue
/*If the card is a face, a number must be subtracted off so it equils 10 or 11 if its an ace*/
_sub4:
sub R1,#3
ldr R0, output_ace
b _addtotal
_sub3:
sub r1,#3
ldr R0, output_king
b _addtotal
_sub2:
sub r1,#2
ldr R0, output_queen
b _addtotal
_sub1:
sub r1,#1
ldr R0, output_jack
b _addtotal

/*Once the suit and face value cards are dealt with, the running card total is determined*/
_addtotal:
/*Add the running total of the cards to the card total*/
add R5, R5,r1
mov R6, R3
bl printf
/*Then the output is given to the user*/
cmp R6, #1
BEQ _h
cmp R6, #2
BEQ _d
cmp R6, #3
BEQ _c
```

```
        b _s
        /*First the card, followed by the suit is outputted*/
        _h:
        ldr r0, output_hearts
        bl printf
        b _loop
        _d:
        ldr r0, output_diamonds
        bl printf
        b _loop
        _c:
        ldr r0, output_cloves
        bl printf
        b _loop
        _s:
        ldr r0, output_spades
        bl printf
        b _loop
        /*Loop until the proper amount of cards are drawn for a given turn*/
        _loop:
        sub r4,#1
        cmp r4,#0
        bgt loop_rand
        /*store the running total back into r1 for main*/
        mov r1, R5

        /*pop back the lr to go back to the main function*/

   pop {lr}
        bx lr

 /*Addresses for each suit and face value card*/
current_card: .word current
output_hearts: .word hearts
output_diamonds: .word diamonds
output_cloves: .word cloves
output_spades: .word spades
output_jack: .word jack
output_queen: .word queen
output_king: .word king
output_ace: .word ace

/*External Functions*/
.global printf
.global time
.global srand
.global rand
```

Dealer Card Function:

```
.data
.balign 4
message: .asciz "My card total is %d\n\n\n"

.balign 4
return4: .word 0

.balign 4
current: .asciz "%d"

.balign 4
hearts: .asciz " of hearts\n"
.balign 4
diamonds: .asciz " of diamonds\n"
.balign 4
cloves: .asciz " of cloves\n"
.balign 4
spades: .asciz " of spades\n"



.balign 4
jack: .asciz "Jack"

.balign 4
queen: .asciz "Queen"

.balign 4
king: .asciz "King"

.balign 4
ace: .asciz "Ace"

.balign 4
card_t: .word 0

.text

.global _dcard
_dcard:
   push {lr}

        mov r4,r0               /* Setup loop counter */
        mov r5, #0
```

```
        mov r0,#0                   /* Set time(0) */
bl time                 /* Call time */
        bl srand                    /* Call srand */

loop_rand:
        bl rand                 /* Call rand */
        mov R1,r0,ASR #1            /* In case random return is negative */
        mov r2,#52

        bl divMod                   /* Call divMod function to get remainder */
        add R1,#2

        cmp R1, #14
        BLE _hearts

        cmp R1, #27
        BLE _diamonds

        cmp R1, #40
        BLE _cloves

        BGT _spades

        _hearts:
        mov r3, #1
        b _continue

        _continue:
        cmp R1,#14
        BEQ _sub4
        cmp R1,#13
        BEQ _sub3
        cmp R1,#12
        BEQ _sub2
        cmp R1,#11
        BEQ _sub1

        ldr R0, current_card
        b _addtotal

        _diamonds:
        mov r3, #2
        sub R1,#13
        b _continue


        _cloves:
        mov r3, #3
```

```
sub R1,#26
b _continue

_spades:
mov r3, #4
sub R1,#39
b _continue

_sub4:
sub R1,#3
ldr R0, output_ace
b _addtotal
_sub3:
sub r1,#3
ldr R0, output_king
b _addtotal
_sub2:
sub r1,#2
ldr R0, output_queen
b _addtotal
_sub1:
sub r1,#1
ldr R0, output_jack
b _addtotal

_addtotal:
/*Add the running total of the cards to the card total*/
mov R5, r1
mov R6, R3

cmp r4, #1
bgt _skip
bl printf

_skip:
cmp R6, #1
BEQ _h
cmp R6, #2
BEQ _d
cmp R6, #3
BEQ _c

b _s

_h:
cmp r4, #1
bgt _loop
ldr r0, output_hearts
```

```
        bl printf
        b _loop
        _d:
        cmp r4, #1
        bgt _loop
        ldr r0, output_diamonds
        bl printf
        b _loop
        _c:
        cmp r4, #1
        bgt _loop
        ldr r0, output_cloves
        bl printf
        b _loop
        _s:
        cmp r4, #1
        bgt _loop
        ldr r0, output_spades
        bl printf
        b _loop

        _loop:
        sub r4,#1
        cmp r4,#0
        bgt loop_rand

        mov r1, R5


   pop {lr}
        bx lr

address_of_message: .word message
address_of_return4: .word return4
cardt: .word card_t
current_card: .word current
output_hearts: .word hearts
output_diamonds: .word diamonds
output_cloves: .word cloves
output_spades: .word spades
output_jack: .word jack
output_queen: .word queen
output_king: .word king
output_ace: .word ace

/*External Functions*/
.global printf
.global time
```

```
.global srand
.global rand
```

DivMod Function (Provided by Dr. Lehr):

```
/*
        Functions
                scaleRight
                addSub
                scaleLeft
                divMod
*/

.text

/*void scaleRight(int &r1,int &r3,int &r2) */
.globl scaleRight
scaleRight:
        push {lr}           /* Push lr onto the stack */
        doWhile_r1_lt_r2:    /* Shift right until just under the remainder */
                mov r3,r3,ASR #1; /* Division counter */
                mov r2,r2,ASR #1  /* Mod/Remainder subtraction */
        cmp r1,r2
        blt doWhile_r1_lt_r2
        pop {lr}            /* Pop lr from the stack */
   bx lr              /* Leave scaleRight */
/* end scaleRight */

/* void addSub(int &r3,int &r2,int &r0,int &r1) */
.globl addSub
addSub:
        push {lr}      /* Push lr onto the stack */
        doWhile_r3_ge_1:
                add r0,r0,r3
                sub r1,r1,r2
                bl scaleRight
        cmp r3,#1
        bge doWhile_r3_ge_1
   pop {lr}      /* Pop lr from the stack */
   bx lr         /* Leave addSub */
/* end addSub */

/* void scaleLeft(int &r1,int &r3,int &r2) */
.globl scaleLeft
scaleLeft:
        push {lr}           /* Push lr onto the stack */
        doWhile_r1_ge_r2:    /* Scale left till overshoot with remainder */
                mov r3,r3,LSL #1  /* scale factor */
```

```
                mov r2,r2,LSL #1  /* subtraction factor */
                cmp r1,r2
        bge doWhile_r1_ge_r2  /* End loop at overshoot */
        mov r3,r3,ASR #1      /* Scale factor back */
        mov r2,r2,ASR #1      /* Scale subtraction factor back */
        pop {lr}            /* Pop lr from the stack */
   bx lr               /* Leave addSub */
/* end scaleLeft */

/* void divMod(int &r2,int &r0,int &r1) */
.globl divMod
divMod:
        push {lr}       /* Push lr onto the stack */
        /* Determine the quotient and remainder */
        mov r0,#0
        mov r3,#1
        cmp r1,r2
        blt end
                bl scaleLeft
                bl addSub
        end:
        pop {lr}      /* Pop lr from the stack */
   bx lr         /* Leave addSub */
/* end divMod */
```