Juichi Lee

December 7, 2019

Package1 Case Studies Analysis

Battleship.asm

Program Description:

The program simulates the game Battleship using assembly. It takes in a number between 0 and 4,294,967,295 and creates a 5x5 board for both the player and computer. Then, it prompts the player to place their 1x1, 2x1, and 3x1 ship on the newly created board. Once the player is done setting up their board, the computer generates random coordinates to place its ships. After the boards are set up and the pieces are placed, the player and computer take turns guessing coordinates on the board to hit the enemy's ships. Between the player and computer's turns, the program displays and checks the state of the board to determine if the player or the computer wins. If either the player or computer scores 5 hits, then the game ends and prompts the player if they want to play again. If yes, then main is called again. Otherwise, the program exits.

Procedures Descriptions:

Intro: Displays welcome message and prompts for a seed number from the player. Then a loop that loops for n times, where n = seed number, generates 0 over and over again using the RandomRange function. This loop appears to have no purpose other than making the player wait for the game to begin.

Fillboard: Takes in an empty board(board2(computer/radar board) or board1(playerboard)) and sets up the board arrays to be 25 x 4 bytes long.

Display: Takes in board1 and board2 and displays their contents. Using two for loops, one for displaying board1 and one for displaying board 2, the procedure checks each board array position to check if it contains a hit(denoted by 5), a miss(denoted by 4), or nothing(denoted by 0). It then prints the symbol representing what was contained in that array position. Strnum is used to keep track of the current position of the element being displayed and is also used by the procedure generate a new row every 5th element.

Plcship functions: Takes in board1, asks the player for coordinates, and places their ship on board1. Each of the Plcship functions are structured similarly. First, it asks how the player desires the piece to be placed and stores their answer as 1(vertical) or 2(horizontal). Then, after jumping to either the horizontal or vertical label, it prompts the player for the row and column they want to place their ship. After the desired row and column are received, the procedure loops to the location of the desired row and column and places a number representing the type of ship(3, 2, 1) at that location. For the 3x1 and 2x1 ships, a nested loop finds the other locations in the array to complete the rest of the ship.

Cplcship functions: Similar to the Plcship functions, however, it takes in board 2 and in the areas where Plcship prompts the player for vertical or horizontal placement and the desired row and column of the ship, it generates a random number using RandomRange instead.

Chckbrd: Takes in board1 and board2 and analyzes their current board state. It first checks if the player has won by looping through board2, adding the number of hits the player has scored against the computer's ships. If the number of hits equals 5, then the player wins. If the player has not scored 5 hits, then it checks if the player has lost by looping through board1 and adding the number of hits the computer has scored against the player's ships. If the number of hits equals 5, then the player loses. If the player does not win or lose, the procedure does nothing and exits. If the player wins, or loses, the procedure displays the appropriate message and prompts the player whether they want to play again.

Plyrtrn: Takes in board2 and places a peg in the location of the player's desired row and column. First, it asks for the player's desired column and row of where they want to place their peg. For both the labels prompting the user for their desired column and row, it checks whether the location the player requested is valid within the 5x5 board. If the location is valid, then procedure checks if a peg is already in that location. If the number stored at the location in the array is a 4 or 5, then the location is invalid. Otherwise, the location is either a miss(0) or hit(1,2,3) depending on the number. Once the procedure confirms that the location is valid, it then changes the number stored at the location into a 4(miss) or 5(hit) based on the previously stored number.

Comptrn: Similar to the Plyrtrn procedure, however, it takes in board1 and randomly generates the row and column it wants to place a peg using Randomrange instead.

How the programmers use registers?

The programmers used the registers EAX and EDX to input values to and store values from Irvine functions, such as WriteString, ReadDec(), and RandomRange. For procedures involving the boards, they used the registers ESP and EBP to retrieve the board arrays pushed onto the stack. For iterating through the board arrays and retrieving the array elements, they used ESI and EDI to represent the board arrays for the player and computer and its location in memory. For looping, they used ECX. EBX was used as an accumulator for creating a new line after every 5th element in the display procedure.

Do they use sub-registers?

It does not appear that the programmers used sub-registers in their Battleship program.

Which addressing modes the programmers use?

They used the register indirect addressing mode for all the procedures involving the boards array.

Do the programmers use procedures?

Yes. Only procedures are called in the main procedure.

How they implement parameter passing?

They passed parameters into their procedures by pushing them onto the stack, and then using esp and ebp to retrieve them within the procedure.

Do the programmers use the stack?

Yes. Before a procedure with board1 or board2 as parameters is called, the offset of board1, board2, or both is pushed onto the stack.

Do the programmers use constants and macros?

The programmers used a single constant, max, to represent the size of the board arrays. The programmers do not use macros in their program.

Do the programmers use the FPU?

The programmers do not use the FPU in their program.

How the programmers handle I/O?

They used the Irvine functions WriteString and WriteDec to display prompts and the board state. They used the Irvine functions ReadDec to read the seed and coordinate information the user inputs.

Are there any procedures or instructions that you did learn in the class?

Yes. All of the Irvine procedures used in the program, the instructions cmp, jmp, inc, and the conditional jump instructions.

Can you figure out what they do?

Yes. (Refer to Procedures Description above)

Are there any graphics?

Yes. The program provides a well structured and intuitive user interface that clearly displays the board state and its game pieces.

How the programmers implement graphics?

The programmers implemented the graphics through their display procedure, which uses loops, special characters stored as global variables, WriteString, and WriteDec to display each board.

Do the programmers include enough documentation?

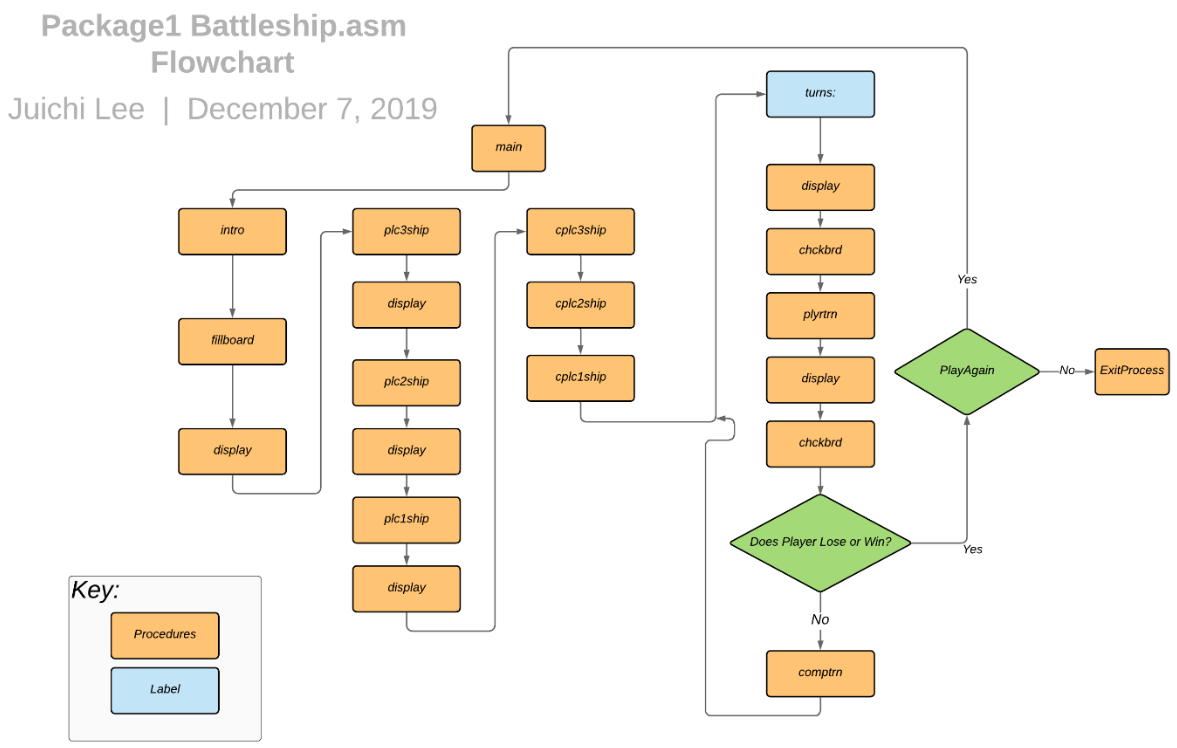
No. The documentation for each procedure only states what it does and provides no information on how it accomplishes its task. Also, it is not immediately clear about what registers each procedure uses and changes. In addition, the documentation for each procedure is missing a pre-condition and post-condition description.

Bugs/Improvements:

Firstly, the seeding functionality in the intro procedure does not appear to serve any purpose given the objective of the program. The RandomRange function used within the runseed loop only returns 0, which is not stored nor used in the program anywhere. I would suggest that the getseed and runseed labels and their code in the intro procedure be removed entirely.

Secondly, there seems to be a typo in the Chckbrd procedure. The typo can be found in the conditional jump label under the addhitl label, which checks if the player has lost. The conditional jump label is JE playerwin where it should be JE playerlose. The bug resulting from this typo makes it so that if the player will never lose against the computer.

Battleship.asm Functional Flowchart:



Snake.asm

Program Description:

This program simulates the game snake.asm in assembly. When the game starts, the player is presented with a GUI. On the interface is a symbol p representing the snake's head, an @ representing the apple, and walls surrounding the snake and apple. Once the player hits one of the directional keys on the keyboard, the game starts and the snake moves in direction of the previously pressed directional key. If the snake eats the apple, then the snake grows by 1 node and a new apple is generated and randomly placed within the game boundary. For every 10 apples the snake eats, the snake's speed increases. If the snake hits its own body nodes or a wall, the program shows the game end screen and prompts the user if they want to try again.

Procedure Descriptions:

EatApple:
Takes in the global variables appleeaten, x_apple, y_apple, x_head, y_head, and score. First the procedure checks if the apple has already been eaten by checking if appleeaten is equal to 1(not eaten), or 0(eaten). If the apple has already been eaten, then the procedure resets appleeaten to 1, creates a new apple by generating random x and y coordinates for it on the screen space using RandomRange and prints it to the screen using Gotoxy and WriteChar. If the apple has not already been eaten, then the procedure checks if it is currently being eaten by comparing the location of the snake's head with the location of the apple. If the locations are not the same, then the program exits. However, if they are, then appleeaten is set to 0, a new node is added to the snake, the game score is updated, and the score is printed to the screen.

AddNodes:
Takes in the global variables NumNodes, Nodes_x, Nodes_Y, x_head, y_head, appleeaten, NumOfNodes, x_apple, and y_apple. If the number of nodes is less than 1, then the procedure adds the x and y location of the snake's head to nodes_x and nodes_y and the procedure exits. However, if the number of nodes is greater than or equal to 1, then the procedure checks if the apple has been eaten. If it was not eaten, then the snake's configuration is updated and the nodes are printed to the screen. If it was eaten, then it starts a loop that loops for how many nodes are in NumOfNodes. The loop shifts all of the node's positions stored in nodes_x and nodes_y to the right by 1 to make space for a new node located at where the snake's head used to be.

Configure:
Takes in global variables Nodes_x, Nodes_y, x_tail, y_tail, and NumOfNodes. Removes the old tail and shifts all of the location values in Nodes_x and Nodes_y to the left to renew the location of each node from the previous shiftright loop change in the AddNodes procedure.

CrashSnake:
Takes in global variables NumOfNodes, Nodes_x, Nodes_y, x_head, y_head, and head. First, the procedure checks if the snake has eaten any of its nodes. It does this by comparing the heads x and y value are compared with the nodes and tail's x and y values in a loop that starts after 3 nodes. If the loop ends without the procedure detecting the head location equaling the location of

a node or tail, then the procedure exits. However, if this is not the case, then the snake's head position and the game over screen is printed to the screen. Then, the procedure asks the player if they want to try again.

GameSpeed:
Takes in global variables score and speed. Calculates and updates the speed of the snake in the game. It calculates the speed by taking the score and dividing it by 10. If the quotient is greater than 1, then the speed is increased by 10. Otherwise, the procedure exits. This makes it so that the speed of the game increases by 10.

How the programmers use registers?

The registers EAX is used for accumulating values, for input and output, and as the parameters for arithmetic instructions. EBX is used as basic pointers to access memory. ECX is used as a counter for loops. EDX is used to temporarily store data in a procedure and outside the procedure. ESI is used to retrieve the offset of arrays in the memory and set values to EBX.

Do they use sub-registers?

Yes. AX, AH, AL, and BX used to initialize the game screen, initialize the values of certain global variables, such as speed and appleeaten, and as parameters in arithmetic instructions. DL DH are used to pass in x and y coordinates to the Gotoxy function, which uses those coordinates to place the cursor on the screen.

Which addressing modes the programmers use?

The programmers use base-indexed addressing modes in all of their procedures that involve processing the Nodes_x and Nodes_y array.

Do the programmers use procedures?

Yes.

How they implement parameter passing?

They pass parameters into procedures by referencing global variables.

Do the programmers use the stack?

No. The programmers do not use the stack since the parameters for procedures are passed by using global variables.

Do the programmers use constants and macros?

No. The procedure does not use constants nor macros.

Do the programmers use the FPU?

No. The programmers do not use the FPU.

How the programmers handle I/O?

The programmers handle input by using the ReadKey function to get the key the player desires to move in and the readchar function to check if the user wants to play again. As for output, they use the function Gotoxy and WriteChar in conjunction to move the cursor to location on the screen and print out a symbol representing the wall, apple, snake, or nodes there. They also use WriteString to print out messages and information about the game's current state.

Are there any procedures or instructions that you did learn in the class?

The procedures and instructions in the procedure that I have learned in class include all of the Irvine library functions, such as WriteString, ReadKey, ReadChar, and WriteChar, and the instructions mov, cmp, jmp, inc, the arithmetic instructions, and the conditional jump instructions.

Can you figure out what they do?

Yes. (Refer to Procedures Description above)

Are there any graphics?

Yes.

How the programmers implement graphics?

The programmers use Gotoxy and the Irvine Library functions WriteString and WriteChar to set up and update the GUI for the snake.asm game.

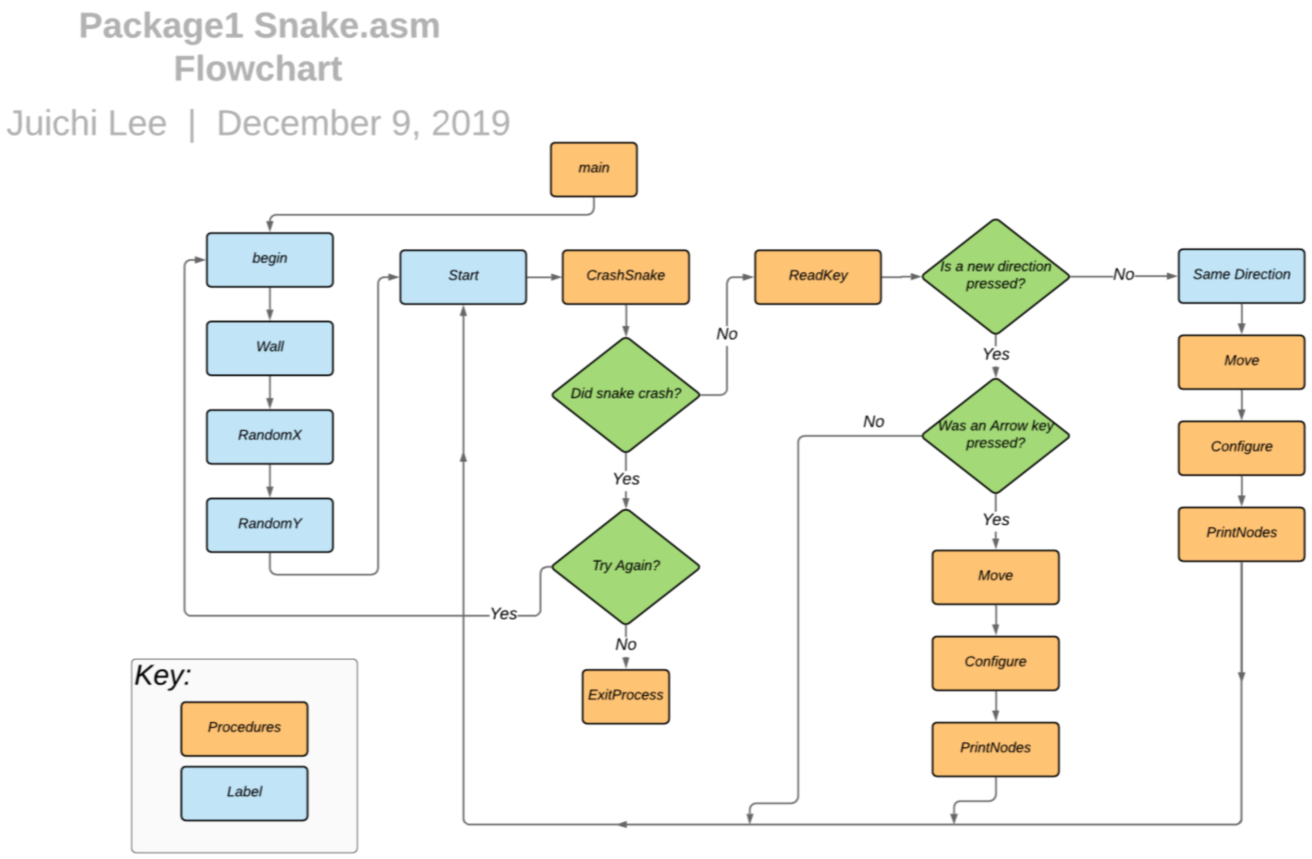
Do the programmers include enough documentation?

Yes. However, they should include documentation for the input and outputs for each procedure since they are using global variables and it's not immediately clear what global variable inputs they are inputting and outputting to. In addition, it appears some of the lines of documentation within each procedure are cut off or hard to understand.

Bugs/Improvements

There were several bugs I noticed while running snake.asm. First, I noticed that after eating 11 apples, a new apple does not appear even though it should. Second, I noticed that the snake's speed increased after eating 1 apple even though it is supposed to increase every 10 apples. Third, after the game ends, and the player tries again, the color from the game over screen(red) is still overlayed onto the new game. Fourth, I noticed that the position of the apple and the player is the same every time the game is first initialized and that the player's initial position never changes despite the program calling for the player's initial position to be randomized every new round.

Snake.asm Functional Flowchart


Package1 Snake.asm Flowchart
Juichi Lee | December 9, 2019
main
begin
Wall
RandomX
RandomY
Start
CrashSnake
Did snake crash?
Yes
No
Try Again?
Yes
No
ExitProcess
ReadKey
Is a new direction pressed?
No
Yes
Same Direction
Move
Configure
PrintNodes
Was an Arrow key pressed?
No
Yes
Move
Configure
PrintNodes
Key:
Procedures
Label