CS 271 Homework 4

Designing and Implementing Procedures

Due Date: 08/04/19

Michael Payne

TA Signature

1 Introduction

Homework 4 asked us to build a procedure-heavy program (meaning, very little work was done in main) that printed a user-defined amount of composite numbers. I thought the assignment was going to be short and easy, but because of some of my design decisions, the opposite ended up being the case. Still, the main requirements of the program along with the extra credit requirements really helped me learn a lot more about passing parameters into a procedure, stack management, and manipulating an array in MASM.

2 Program Initialization, Internal Register, Definitions, and Constants

In the .data section, "intro1", "intro2", "intro3", "extCr1", "extCr2", "extCr3", "extCr4", "instr1", "instr2", "goodbye", "errorMsg", and "programTime", are all strings that are given a byte for each character. An array named "primeList" is assigned 720 bytes (dword with a length of 180). "printCount", "lineCount", and "primeLimit" are each assigned 4 bytes and they all have values of 0. "upperLimit" is assigned 4 bytes and has a value of 920000. "compCount" and "timeStamp" are assigned 4 bytes and are both uninitialized. "mainNum" and "squarePointer" are each assigned 4 bytes and both have a value of 4. "currentSquare" is assigned 4 bytes and has a value of 9. "arrayPosition" is assigned 4 bytes and has a value of 8

3 Main Program

Figure 1 shows all of the main components and functional logic of the program. The "Introduction" segment starts by executing the rdtsc instruction and storing a timestamp into a location in memory called "timeStamp". After that, the program prints its title and my name. The program then prints out all of the extra credit messages (which list the extra credit portions of the assignment that I did).

The program then transitions to the "Get info" segment, which calls the "getInfo" procedure (it passes the arguments "compCount", "instr", "errorMsg", and "upperLimit" through the stack to the procedure). This procedure prints instructions asking the user to choose how many composite numbers they want to be displayed by inputting a number in the range of 1 to 92,000. It takes the user's inputted number and then calls the "validate" procedure. This procedure checks to see if the inputted number is within the previously stated range. If it isn't, it prints an error message and prompts the user to input another number. If the user inputs an acceptable number, the program returns to the "getInfo" procedure, and user's input is returned or outputted by copying it to the "compCount" memory location.

The program then moves on to the "Array setup" segment, which calls the "arraySetup" segment. This segment takes the address of array "primeList" as an argument and preps it for the next segment by inserting the values 2 and 3 into the first two positions in the array.

Once the procedure has done this, it returns to main, and the program progresses to the "Calculating and printing composites segment". This segment pushes the value of "primeLimit", the address of "primeList", and the values of "printCount", "lineCount", "compCount', "arrayPosition", "squarePosition", "squarePointer", "mainNum", and "currentSquare" to the stack (so they can be passed as arguments to "showComposites").

It then calls the "showComposites" procedure, which begins the lengthy process of calculating and printing the user-defined number of composite numbers. "showComposites" starts by setting up the stack frame and then moving the address of the stack to esi. It then transitions to "squareCheck", which compares the current value in "mainNum" to the current value in "currentSquare". If "mainNum" is lower, the program jumps to "cmpCheckInit". If "mainNum" is higher, "squareCheck" increments "squarePointer" by 4 (to point to the next value in "primeList") and then copies the next value in "primelist" to eax. It squares this value and copies the result to the location in the stack where "currentSquare" is stored. Since the program now has a new "currentSquare that is higher than "mainNum", it immediately jumps to "cmpCheckInit" to begin checking the current value in "mainNum" to see if it is a prime or composite number.

"cmpCheckInit" assigns 0 to ecx and then transitions to the "cmpCheck" loop. "cmpCheck" starts at the beginning of the "primeList" array and copies a prime number. It the divides "mainNum" by the selected prime number. If the remainder of the division is 0, the program jumps to "isCompositeCall". If the remainder isn't equal to zero, the program then checks to see if the currently selected prime number is equal to the prime number that "squareNumber" points to. If it is equal, the program jumps to "primeAdd", because the value in "mainNum" is a prime number and needs to be added to the "primeList" array. If the two numbers aren't equal, ecx is incremented by 4 (so that the next number in "primeList" can be used to divide the value in "mainNum"), and the loop starts over. This is continued until the number is determined to be either prime or composite.

Again, if "mainNum" is determined to be prime (we'll move on to what happens if the number is composite after "primeAdd" is described), the program jumps to "primeAdd". This block of code first checks to see if "primeLimit" is true or false. If it is true, that means that the program has hit the limit of prime numbers that can be inserted into "primeList" (the array can only hold 180 numbers), and "primeAdd" will no longer execute (as there is no reason to insert prime numbers into the array anymore). The program instead jumps to "restart" where "mainNum" is incremented by 1 so that "showComposite" can check a new number. If "primeLimit" is false, "primeAdd" copies the value in "mainNum" to the "primeList" array and then increments "arrayPosition" to point to the next empty spot in the array (so that the program knows where to insert the next prime number). "mainNum" is then incremented by 1 and "primeAdd" then checks to see if the last prime number stored in "primeList" is equal to the prime number 1013. If it is, "primeLimit" has its value changed to true, so that "primeAdd" will no longer run. The program then jumps to "isCompositeCall" so that the next number can be printed (the program increments "mainNum" in such a way that even numbers (except for 4) are never checked and are automatically printed).

When the program moves to "isCompositeCall", it pushes "printCount", "lineCount", and "mainNum" to the stack (to be passed as parameters to the procedure "isComposite"), and the procedure "isComposite" is called. This procedure prints the passed-in "mainNum" value and then prints a number of spaces (till the next composite number in the line) that are based off the number of digits in "mainNum". The formula is (9 - number of digits to right of the first digit) + 5. This allows the program to print the composite numbers in neatly aligned columns. "isComposite" then increments "printCount" by 1 and then checks its value against the number 6. If it is less than 6, it jumps to "printDone" and the procedure ends. If it is equal to 6, it prints a line break, sets "printCount" to 0 and then starts working on "lineCount". "lineCount" is incremented by 1 and then checked against the number 10. If it is less than 10, the program jumps to "printDone" and the procedure ends. If it is equal to 10, the program pauses and waits

for an enter key input from the user before it continues. “isComposite” then sets “lineCount’ to 0, and the program returns to “showComposite”.

Once the program returns to “showComposite”, “compCount” is decremented, and the program jumps to “compNumLoop”. If “compCount” equals 0, the program jumps to finish and the procedure ends. If it is greater than 0, the program moves to “restart” where “mainNum” is incremented by 1. “restart” then checks to see if the new “mainNum” is even. If it is, the program immediately jumps to “isCompositeCall” where the new “mainNum” is printed. If “mainNum” is odd, the program jumps to “squareCheck” where the sequence of checks to see if “mainNum” is prime or composite start up again.

“finished” is the final block of code in “showComposite”. The stack is restored, and the procedure ends.

After printing the required number of composite numbers, the program moves on to the “Goodbye” segment. This segment prints out a goodbye to the user and then uses rdtsc to generate a new time stamp. The old timestamp is subtracted from the new one, and the result is printed (this is the execution time of the program). Finally, the program ends.

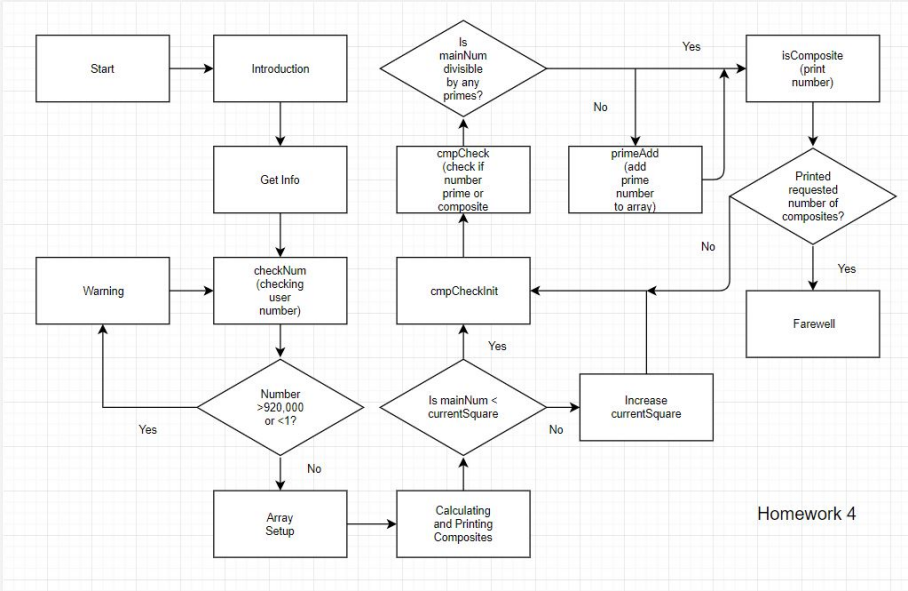

Figure 1: Block Diagram for Homework 4

4 Procedures

All of these procedures are covered in great detail in “Main Program” and in the comments in my code, so I will be very brief in my descriptions of them.

1. “introduction”

   “introduction” is called by segment 1 in order to print introductory text that lists the title of the program and my name.

2. “getInfo”

   The “getInfo” procedure gives brief instructions about how the program works to the user and then takes a number inputted by the user and passes it to the “validate” procedure. It

then takes the returned number from "validate" and copies it to the "compCount" memory location.

3. "validate"

   "validate" checks inputted number passed in by "getInfo" and checks to see if it falls in the previously defined range (1 to 92,000). If it fails the check, "validate" prints an error message and prompts the user to enter another number. If it passes the check, the number is returned to "getInfo"

4. "arraySetup"

   "arraySetup" preps "primeList" so that it can be used by "showComposites". It does this by inserting a 2 and a 3 into the first 2 locations in the array.

5. "showComposites"

   "showComposites" checks numbers to see if they are prime or composite. If they are prime, they are added to the "primeList" array. If they are composite, "isComposite" is called, and the number is printed. After a number is printed, "compCounter" (that is determined by the number inputted by the user in "getInfo) is decremented. If the newly decremented number equals 0, the procedure ends and returns to main. If it isn't 0, the procedure moves to "reset" where "mainNum" is incremented by 1 (and then checked to see if it is even; even numbers are automatically printed and not checked since they are always composite). The procedure then restarts and the newly incremented "mainNum" is checked to see if it is prime or composite.

6. "isComposite"

   "isComposite" takes in a passed in composite number ("mainNum") and prints it out. It then prints uses the formula (9 - digits to right of first digit)+ 5 to print out the number of spaces in between the just printed composite number and the next or upcoming composite number. This makes it so the printed numbers are in aligned columns. "isComposite" prints 6 composites per row. After printing 10 rows, the procedure pauses the program and waits for the user to input an enter value before unpausing the and continuing the program.

7. "farewell"

   "farewell" prints a farewell message to the user.

5 Stored Program Data

1. "Introduction" stage

   (a) intro1 = "Composite Numbers by Michael Payne"

   (b) intro2 = "Tell me how many composite numbers you would like to see."

   (c) intro3 = "I can show up to 920,000 composites."

   (d) extCr1 = "**EC: This program aligns the output columns."

   (e) extCr2 = "**EC: This program displays more composites (and prints a page at a time)."

(f) extCr3 = "**EC: This program uses an array to make checking for composites more efficient."

(g) extCr4 = "**EC: This program calculates and prints its execution time."

(h) instr1 = "Enter how many composites you want to display."

(i) instr2 = "The number must be ¿= 1 and ¡= 920,000."

(j) goodbye = "Thank you for using my program!"

(k) errorMsg = "Out of range. Try again."

(l) programTime = "The elapsed program time is: "

(m) compCount = ?

(n) upperLimit = 920000

(o) mainNum = 4

(p) primeList = 180 DUP(?)

(q) currentSquare = 9

(r) squarePointer = 4

(s) arrayPosition = 8

(t) printCount = 0

(u) lineCount = 0

(v) primeLimit = 0

(w) timeStamp = ? = some random time stamp that is taken from eax (which is generated by rdtsc)

2. "Get Info" stage

   (a) compCount = 10

3. "Array setup" stage

   (a) primeList = 2 and 3 are inserted into first 2 indices of primeList

4. "Calculating and printing composites" stage

   (a) primeList = Since the user is only asking for 10 composites to be displayed, the prime numbers 5, 7, 11, 13, and 17 will be inserted into primeList

5. "Goodbye" stage

   (a) timeStamp = new timestamp is generated and old one is subtracted from it; let's say it is 10000

6 Difficulties and Concluding Thoughts

I think this assignment would have been fairly simple if I had stuck to the basic requirements and also used global variables instead of passing in parameters via the stack when calling procedures. The combination of those 2 factors along with headache of updating and accessing the "primeList" array upped the complexity and difficulty of the assignment by a huge amount.

As stated before, I decided to pass in parameters through the stack instead of utilizing global variables. Although I did genuinely learn a lot, I do regret this decision, because it made finishing the assignment take far longer than I had anticipated. One of the biggest issues that I faced was debugging became a nightmare when I finally got to "showComposite". Because I had passed in so many parameters, "showComposite" was constantly having to access and manipulate data on the stack. In addition, "showComposite" had a large number of complicated moving parts, and because of the way I had designed the program (specifically, how I chose to set the upper limit to 920,000), it had to be able to run thousands of times without breaking down. And, of course, things did break down and zeroing in on the various problems was a torturous experience.

The only thing that could have made this manageable, I think, would have been if had broken "showComposite" down into extremely tiny parts. I would then have been able to test each individual component thoroughly and gradually link everything together (while stopping and continuing to test the various combinations of segments). Although my actual approach did involve breaking "showComposite" into individual pieces, I think I was too hasty when merged everything together (also, the components were too large). The result was that when I joined everything together, everything broke down horribly. And when I would fix an issue by making some adjustments, something else would inevitably break in response.

I think the issue is that when I'm reading through the homework requirements, everything always seems easier than it actually is, and that tricks me into never doing the necessary amount of preparation.

7 Extra Credit

This program completes all of the extra credit tasks (well, with the exception of the one asking me to fine-tune assignment 2; I wasted so much time on this assignment that I never got to that).

Calculating the program execution time and getting the composite numbers to display in aligned columns were both fairly easy things to code, since I had done that in previous assignments (I simply cannibalized code and ideas from the assignments that had me do those things). The other 2 requirements weren't terribly difficult if looked at and approached in a vaccuum, but the combination of managing an array and increasing the number of composite numbers displayed (which meant more iterations for "showComposites") along with my decision to pass in parameters to my procedures via the system stack added so much complexity (at least for someone with my level of programming knowledge and skill) to the assignment. I had to keep track of the end of the array (where I was sticking newly discovered prime numbers), and also where the source of "currentSquare" was located, and this gave me a fair amount of trouble (again, a lot of this was a byproduct of my struggle with stack management).

The silver lining to all of this is that I do think I genuinely learned a lot from this assignment. Hopefully, it makes the final project a little easier.

8 Source Code

```
TITLE Homework4 (Michael_Payne_hw4_code.asm)

; Author: Michael Payne
; Last Modified: 08/03/2019
; OSU email address: paynemi
; Course number/section: cs271
; Project Number: Homework 4 Due Date: 08/04/2019
; Description: This program calculates and displays composite numbers. The
   program first lists the program title and my name.
; It then tells the user to enter a number in the range of 1 to 920,000 (this
   input specifies how many composites will be displayed).
; Numbers less than 1 and greater than 920,000 will produce an error message.
   Before the program starts calculating and displaying
; composite numbers, it preps the primeList array by inserting a few values. The
   program then moves on to the next segment and starts
; calculating and displaying composite numbers (the specifics of this are
   complicated and go beyond the scope of this introduction - I will
; elaborate on this in later comments). After it displays the specified number of
    prime numbers, the program says goodbye to the user, and
; calculates and dispays the execution time of the program.

INCLUDE Irvine32.inc



.data
intro1   byte "Composite Numbers by Michael Payne",10
intro2   byte "Tell me how many composite numbers you would like to see.", 10
intro3   byte "I can show up to 920,000 composites.", 0
extCr1   byte "**EC: This program aligns the output columns.", 10
extCr2   byte "**EC: This program displays more composites (and prints a page at
   a time).", 10
extCr3   byte "**EC: This program uses an array to make checking for composites
   more efficient.", 10
extCr4   byte "**EC: This program calculates and prints its execution time.", 0
instr1   byte "Enter how many composites you want to display.",10
instr2   byte "The number must be >= 1 and <= 920,000.",0
goodbye   byte "Thank you for using my program!",0
errorMsg  byte "Out of range. Try again.",0
programTime  byte "The elapsed program time is: ", 0
compCount  dword ?   ; user inputted number of composite numbers to be displayed
upperLimit  dword 920000  ; max number that user can input for displayed
```

```
   composite numbers
mainNum   dword 4   ; number that program is running checks on to see if it is
   prime or composite
primeList  dword 180 DUP(?) ; array that stores prime numbers. Only stores up to
   1013
currentSquare dword 9   ; squared prime number that limits determines how many
   primes the value in mainNum will be divided
squarePointer dword 4   ; keeps track of which prime number in primeList is
   currently being squared
arrayPosition dword 8   ; keeps track of last prime number that has been added to
    primeList
printCount  dword 0   ; counter for how many numbers have been printed (program
   jumps to new line after 6 printed numbers)
lineCount  dword 0   ; counter for how many lines have been printed (program
   pauses at 10th line)
primeLimit  dword 0   ; boolean value essentially - is changed to 1 after 1013
   has been added to primeList
timeStamp  dword ?


.code
main PROC
; segment 1 - Introduction - This segment introduces the program. It gives the
   title of the program and my name. It then explains that this program
; will list a number of composite numbers specified by the user. It calls the
   introduction procedure and passes in the the address of the intro1
; variable. Afterwords, it manually ESP back to its original address since it isn
   't necessary to pop anything.
rdtsc
mov   timeStamp, eax
push  offset intro1  ; passing address of intro1 as argument (which results in
   intro2 and intro3 being printed as well)
call  introduction  ; calling introduction procedure
add   esp, 4    ; adjusting stack to original position

; EXTRA CREDIT SEGMENT - This lists the extra credit parts of the assignment that
    I have completed
mov edx, offset extCr1
call writestring
call crlf

; segment 2 - Get info - Segment 2 instructs the user to input a number in the
   range of 1 to 920,000 (this input will determine
; how many composite numbers are going to be displayed. It calls the getInfo
   procedure, and passes addresses of compCount, instr1
; (which results in instr2 being printed as well), and errorMsg in as arguments.
```

```
   compCount is passed by reference
; (as an output parameter), because the user's input will become its value.
   upperLimit is passed in as an input parameter because we need to use its
   value for
; validation of the user's input. After printing instructions, it takes a number
   from the user as input and then checks it against upperLimit and the integer
   1.
; Anything greater than upperLimit and less than 1 will produce an error message
   and a prompt to enter another number. Once an acceptable number is inputted,
; it is stored at compCount's address, and the program returns to main. The
   program then moves the stack back to its original position (since nothing
   needs to be
; popped).
push  offset compCount ; need to assign user's inputted number as compcount's
   value, so it is being passed in as an output paramter
push  offset instr1  ; passing address of instr1 so instructions can be printed
push  offset errorMsg  ; passing address of errorMsg in case user inputs number
   outside of accepted range
push  upperLimit   ; passing in value as input parameter, because we need it for
   error checking
call  getInfo    ; calling getInfo procedure to get user input and assign it to
   compCount
add   esp, 12    ; manually adjusting where ESP is pointing to

; segment 3 - Array setup - Array Setup preps primeList (the array where prime
   numbers will be stored) for segment 4 by inserting the values 2
; (inserting 2 was a mistake, but I don't want to reconfigure my program at this
   point) and 3 into it. The only argument passed in is the address
; of primeList (so that the 2 aforementioned values can be inserted into the
   array). After adjusting the array, the procedure returns to main,
; and ESP is changed to its original position.
push  offset primeList ; passing in address of primeList so it can have 2 and 3
   inserted into it
call  arraySetup   ; calling arraySetup to prep primeList
add   esp, 4     ; reverting ESP to its original position
; segment 4 - Calculating and printing composites - Essentially everything that
   this segment does is through the showComposites procedure. Because of how I
   type
; my comments, and because of the way the assignment report is structured, I
   often find myself describing the same processes 4-5 times. It would be too
   painful
; to try to do that with showComposites, so my description of segment 4 is going
   to be very brief (since I'm going to describe its function in
; the description of showComposites). Segment 4 passes primeLimit, printCount,
   lineCount, compCount, arrayPosition, squarePosition, squarePointer, mainNum,
; and currentSquare as input parameters to showComposites Their values are needed
```

```
   , but they don't need to be modified in anyway. The address of primeList
; is passed in, because it needs to be both heavily modified and read from.
; Segment 4 uses showComposites to calculate and print out composite numbers (the
    amount is determined by the compCount). Once showComposites is finished,
; the proram returns to main. ESP is then returned to its original position with
   the add instruction.
push  primeLimit   ; boolean value that determines whether prime numbers will be
   added to primeList or not
push  offset primeList ; address for primeList is passed in so that it can be
   used to store prime numbers
push  printCount   ; number of digits that have been printed in a row. Once 6 are
    printed, program prints a line break
push  lineCount   ; Number of lines that have been printed. Once 10 are printed,
   program pauses and asks user to hit enter to continue
push  compCount   ; number of composite numbers that are to be printed
push  arrayPosition  ; this points to the spot in primeList where the program
   inserts new prime numbers
push  squarePointer  ; this points to the prime number that is currently being
   squared (this will be further explained later)
push  mainNum    ; Current number that is being checked/processed by
   showComposites
push  currentSquare  ; square of prime number being pointed to by square pointer
   (in primeList array)
call  showComposites  ; calling showComposites to begin calculating and printing
   composite numbers
add   esp, 32    ; reverting esp to original position
; segment 5 - Goodbye - This segment prints a goodbye message to the user by
   calling the farewell procedure. It passes in the address of goodbye
; to accomplish this. Once farewell is finished, the program returns to main and
   adjusts the where ESP is pointing to. The program then calculates
; its execution time with rdtsc and prints out the result.
call  crlf    ; line break for readability
push  offset goodbye  ; passing in address of goodbye to farewell
call  farewell   ; calling farewell to print goodbye message
add   esp, 4    ; adjusting position of ESP
call  crlf
mov   edx, offset programTime
call  writestring
rdtsc
sub   eax,timeStamp
call  writedec


exit
main ENDP
```

```
; Introduction Procedure - This procedure prints out an introduction message for
   the user at the start of the program
; Implementation Note: Takes an address pushed to stack and uses writestring to
   print it
; receives: address of intro1 string
; returns: nothing
; preconditions: nothing
; registers changed: edx
introduction PROC
push  ebp     ; setting up stack frame
mov   ebp, esp
mov   edx, [ebp+8]  ; moving passed in address to edx
call  writestring   ; printing passed in parameter
call  crlf    ; line break for readability
pop   ebp     ; restoring stack
ret        ; returning to main
introduction endp

; getInfo procedure - This procedure takes in an inputted number from the user
   and then runs it through the validate procedure
; to make sure it falls within the range of 1 to 920,000. It take the value
   returned from validate (stored on the stack), pops
; it into eax, and then assigns it to a paramter passed in by reference (in this
   case, the compCount variable)
; Implementation Note: This procedure accesses its passed in parameters via the
   stack (it establishes a stack frame).
; Parameters must be passed in this order and fashion:
; 1 - return variable - addressed must be passed in, so user input can be copied
   to it
; 2 - instructions string - addressed must be passed in (can result in multiple
   lines being printed if set up that way in .data)
; so that procedure can utilize writestring
; 3 - error string - address must be passed in, so that called validate procedure
    has an error message to use writestring on
; 4 - upperLimit (must have an integer value) value passed in - needs to pass
   this to validate procedure to make sure user's inputted
; value is within program's defined range
; receives: compCount address, instr1 string address, errorMsg string address,
   upperLimit value all on stack
; returns: user input value assigned to address of compCount
; preconditions: upperlimit must be > 1
; registers changed: eax, ebx, edx
getInfo  PROC
push  ebp     ; setting up stack frame
mov   ebp, esp
```

```
mov   edx, [ebp+16]  ; Printing instructions
call  writestring
call  crlf
call  readint    ; getting number from user
push  [ebp+12]   ; passing address of error message to validate
push  [ebp+8]    ; passing upperLimit value for error checking to validate
push  eax     ; passing inputted user number to validate for error checking
call  validate   ; calling validate to make sure inputted user number falls
   within defined range
pop   eax     ; taking value returned from validate and moving it to eax
mov   ebx, [ebp+20]  ; assigning value in eax to return variable (compCount)
mov   [ebx], eax
add   esp, 8    ; moving stack to original position
pop   ebp     ; more stack bookkeeping
ret        ; returning to main
getInfo endp

; validate procedure - This procedure takes a passed in number and makes sure it
   is within the range of 1 and a passed in upperLimit value
; (that needs to be higher than 1). If the number is lower than 1 or higher than
   the upperLimit value, an error message is printed and the user
; is prompted to enter another value. This is continued until the user inputs an
   acceptable value
; Implementation Note: This procedure accesses its parameters via the stack.
; Parameters must be passed in via this order and fashion:
; 1 - error message string - address must be passed in so procedure can use
   writestring
; 2 - upper limit value - a positive number greater than 1 must be passed in for
   error checking
; 3 - number value - a number value must be passed in (I guess it has to be a
   dword) - program will validate it and return it if it falls within defined
   range
; receives: errorMsg string address, upperLimit value, user input value, all on
   stack
; returns: validated user input value on stack
; preconditions: upper limit value must be greater than 1
; registers changed: eax and edx
validate PROC
push  ebp     ; setting up stack frame
mov   ebp, esp
mov   eax, [ebp+8]  ; moving inputted user number to eax to begin validating it
jmp   checkNum   ; jumping to checkNum to begin validating number
; warning is jumped too if the inputted user number doesn't fall within the
   defined range. It prints out an error message, and then prompts the user to
; input another value.
warning:
```

```
mov   edx, [ebp+16]  ; printing error message
call  writestring
call  crlf
call  readint    ; getting user input
; checkNum compares inputted number against 1 and some sort of upper limit. If
   the value doesn't fall within this range, it jumps to warning and gets new
   input.
; If user input passes checks, the value is pushed to the stack, and the program
   returns to getInfo
checkNum:
cmp   eax, [ebp+12]  ; comparing input to upper limit
jg   warning    ; if greater than, jump to warning
cmp   eax, 1    ; comparing input to 1
jl   warning    ; if less than, jump to warning
mov   [ebp+8], eax  ; passed checks, so pushing to stack (as return value)
pop   ebp     ; restoring stack
ret
validate endp

; arraySetup procedure - This procedure takes a passed in array and inserts the
   values 2 and 3 into the first 2 positions. In the context of this program
; , arraySetup is prepping primeList to be used in the showComposites procedure.
; implementation note: the address of an array that consists of dwords with a
   length of at least 2 must be passed into this procedure
; receives: the address of the primeList array in stack
; returns: nothing (it modifies the contents of the array, though)
; preconditions: array must have a length of at least 2 (and it has to be dwords)
; registers changed: ecx, ebx, edi
arraySetup PROC
push  ebp     ; setting up stack frame
mov   ebp, esp
mov   edi, [ebp+8]  ; moving address of array to edi
mov   ecx, 2    ; setting up counter for loop
mov   ebx, 1    ; ebx will store values that are incremented and stored in first
   two indices of array
; arrayLoop is a loop that inserts 2 and 3 into the first two indices of passed
   in array
arrayLoop:
inc   ebx     ; incrementing ebx
mov   [edi], ebx   ; copying value in ebx to selected position in array
add   edi, 4    ; incrementing edi by 4 so that it points to next position in
   array
loop  arrayLoop   ; restarting loop
pop   ebp     ; restoring stack
ret
arraySetup ENDP
```

```
; showComposites procedure - showComposites calculates and prints (well, a called
    procedure in it handles the printing) a user-specified number of composite
; numbers.
; implementation note: This procedure accesses its parameters via the stack.
; parameters must be passed in this order and in this fashion:
; 1 - primeLimit value - a value that must be 0 when passed in. This is used as a
    boolean that is switched to true after showComposite inserts
; 1013 into primeList (which allows the program to find composite numbers up to
   920,000). Once it is switched to 1, stops inserting prime numbers into
; primeList
; 2 - primelist array - address of a dword array that can hold a minimum of 180
   elements - prime numbers are stored and accessed in this array
; 3 - printCount value - the value must be 0, and it is used to keep track of the
    number of composite numbers that have been printed in a row -
; this is primarily passed in to reserve a spot on the stack for a value that
   only needs to be persistant for as long as showComposites is open
; 4 - lineCount value - the value must be 0, and it is used to keep track of the
   number of lines that have been printed in a row - this is like
; printCount in that this is mostly to reserve a spot on the stack for a value
   that needs to be persistant while showComposites is open
; 5 - compCount value - this is the number that the user inputted earlier, and it
    is used to determine how many composite numbers will be found and printed
; 6 - arrayPosition value - this value allows showComposites to know where to
   insert newly found prime numbers in the primeList array
; 7 - squarePointer value - this value allows showComposites to keep track of
   which prime number it is using in primeList to generate currentSquare
; 8 - mainNum value - starts at 4 and is incremented by 1 everytime
   showComposites fully processes a number (whether it's found the number to be
   prime or composite).
; 9 - currentSquare value - showComposites determines whether a number is prime
   or composite by dividing it by a set number of prime numbers that have been
   stored in
; primeList. For example, if 13 is the prime number pointed to by squarePointer,
   it is squared and the result (169) is copied to currentSquare. While the
   currentSquare
; is 169, showComposites will only divide mainNums that it is processing by prime
    numbers lower than 13 when it is determining whether a number is composite
   or not.
; When mainNum (the number that showComposites is always working on) is found to
   be equal to currentSquare (in our example, 169), squarePointer increments to
   the next prime
; number in the list. That number is squared and then copied into currentSquare.
   showComposites then continues to check numbers until mainNum equals
   currentSquare again.
; receives: primeLimit value, primeCount value, lineCount value, compCount value,
```

```
    arrayPosition value, squarePointer value, mainNum value, currentSquare value
    , and
; the address of primeList array all on the stack
; returns: Nothing, although it heavily modifies the primeList array and prints
    composite numbers. it is like a void function
; preconditions: lineCount and printCount must have values of 0. currentSquare
    must be initialized to 9. mainNum, must be initialized to 4. compCount's
    value must
; have been chosen by the user and also fall within the range of 1 to 920,000.
    primeList must have 2 and 3 already inserted into it and also have a length
    of 180 (and
; consist of dwords). squarePointer must be initialized to 4, and arrayPosition
    must be initialized to 8. primeLimit must be initialized to 0.
; registers modified: eax, ebx, ecx, edx, and esi
showComposites PROC

push  ebp     ; setting up stack frame
mov   ebp, esp
mov   esi, [ebp+36]  ; moving address of stack to esi
; squareCheck checks to see if the value stored in mainNum is lower than the
    value in currentSquare. If it isn't, squarePointer is incremented to
; the next prime number stored in primeList, and a new currentSquare is generated
    (by squaring the number that squarePointer points to).
squareCheck:
mov   eax, [ebp+12]  ; moving mainNum to eax
cmp   eax, [ebp+8]  ; comparing mainNum to currentSquare
jl   cmpCheckInit  ; if mainNum less than currentSquare, start check to see if
    prime or composite
mov   ebx, [ebp + 16]  ; moving value of squarePointer to ebx
add   ebx, 4    ; incrementing squarePointer by 4 (to point to next prime number
    in primeList)
mov   [ebp+16], ebx  ; copying new value to squarePointer's location in stack
mov   eax, [esi+ebx]  ; using squarePointer to copy next prime in primeList to
    eax
mov   ecx, eax   ; getting ready to square new prime num
mul   ecx     ; squaring prime num
mov   [ebp+8], eax  ; copyin new value to currentSquare's location in stack
mov   eax, [ebp+12]  ; copying mainNum value back into eax to prep for jump to
    cmpCheckInit
jmp   cmpCheckInit  ; starting process of checking mainNum to see if it is prime
    or composite
; primeAdd takes the value in mainNum (which has been found to be a prime number)
    and adds it to the primeList array. It first checks the primeLimit to see
; if it has a value of true or not. If it is true (value of 1), it jumps to
    restart, and skips the process of adding a prime number to the primeList
    array.
```

```
; This is because the array is only designed to hold prime numbers up to the
   value of 1013. If primeLimit is 0, mainNum is added to primeList, and
   arrayPosition
; is incremented to the next position in primeList. mainNum is then incremented
   by 1, and the prime number that was just added to primeList is checked
; against the value 1013. if it isn't equal, the program jumps to isCompositeCall
   . If it is equal, primeLimit is changed to 1, so that prime numbers are no
; longer added to primeList and the program the program then jumps to
   isCompositeCall (even numbers are not checked as they are all composite, so
; they are simply printed)
primeAdd:
mov   edx, [ebp+40]  ; copying primeLimit value to edx to see if it is true
cmp   edx, 1    ; comparing primeLimit value to 1
je   restart    ; if it is 1, no longer add prime numbers to primeList array.
   jump to restart and increment mainNum by 1
mov   edx, [ebp+20]  ; copying arrayPosition value to edx
mov   ecx, [ebp+20]  ; copying value of arrayPosition to ecx, because prime
   number that is added must be checked at end of primeAdd
add   [esi+edx], eax  ; adding current mainNum (determined to be prime number) to
    primeList
add   edx, 4    ; incrementing arrayPointer to next position in array
mov   [ebp+20], edx  ; copying new value to arrayPointer's position in stack
mov   eax, 1
add   [ebp+12], eax  ; incrementing mainNum's value in stack
mov   eax, [ebp+12]  ; moving new mainNum to eax
mov   ebx, [esi+ecx]  ; checking value of highest prime number in array to see if
    limit has been hit
cmp   ebx, 1013   ; comparing to 1013
jne   isCompositeCall  ; if not equal, jump to print mainNum (since it is an even
    number and is composite)
mov   edx, 1    ; if equal, change primeLimit to true
mov   [ebp+40], edx
jmp   isCompositeCall  ; jumping to isCompositeCall to print mainNum
; cmpCheckInit gets the program ready to run cmpCheck by setting ecx to 0
cmpCheckInit:
mov   ecx, 0    ; copying 0 to ecx so we can begin moving through primeList
; cmpCheck checks the value in mainNum to see if it is a composite or prime
   number. It divides mainNum by a set number of prime numbers that are found in
    primeList
; If the result of the division produces a remainder of 0, the value is
   determined to be a composite number, and the program immediately jumps to
   isCompositeCall
; where the value is printed, and mainNum is eventually incremented by 1. If the
   division produces a remainder that isn't 0, ecx is incremented by 4, and
   mainNum is
; divided by the next prime number found in primeList. This continues until the
```

```
   prime number that is pointed to equals the prime number that is pointed to by
    squarePointer
; . If this happens, the program jumps to primeAdd, and the value in mainNum is
   added to primeList.
cmpCheck:
xor   edx, edx   ; clearing edx for division
mov   ebx, [esi + ecx] ; moving current prime number to ebx
div   ebx     ; dividing mainNum (currently in eax) by selected prime numbber
mov   eax, [ebp+12]  ; copying mainNum back into eax
cmp   edx, 0    ; comparing remainder of previous division to division
je   isCompositeCall  ; if 0, we have a composite number and need to print it,
   getting ready to call printing procedure
mov   edx, [ebp+16]  ; moving square pointer to edx
mov   ebx, [esi+edx]  ; moving value at squarePointer in primeList to ebx
cmp   ebx, [esi+ecx]  ; if number at squarePointer is equal to currently selected
    prime number, we have divided main num by all available
je   primeAdd   ; jump to primeAdd to add prime num
add   ecx, 4    ; incrementing ecx by 4 to use next prime number in primeList
jmp   cmpCheck   ; otherwise, restart cmpCheck process
; isCompositeCall calls isComposite, and passes the values of printCount,
   lineCount, and mainNum as arguments. isComposite prints composite numbers for
    the program.
; it returns the adjusted lineCount and the adjusted printCount back to
   showComposites. the adjusted values are copied to their locations on the
   stack
; and mainNum is stored in eax where it is later incremented in reset.
   isCompositeCall also plays a part in making sure that showComposite only
   prints the number
; of composites that the user specified earlier by starting the process of
   decrementing compCount
isCompositeCall:
push  [ebp+32]   ; pushing printCount to stack
push  [ebp+28]   ; pushing lineCount to stack
push  eax     ; pushing mainNum
call  isComposite   ; calling isComposite so that composite number can be printed
pop   eax     ; storing returned mainNum in eax
pop   ebx     ; storing returned adjusted lineCount in ebx so that it can be
   copied to its position in stack
pop   edx     ; storing returned adjusted printCount in edx so that it can be
   copied to its position in stack
mov   [ebp+28], ebx
mov   [ebp+32], edx
mov   ecx, [ebp+24]  ; moving value of compCount to ecx so that it can be
   decremented
loop  compNumLoop   ; decrementing compCount value and jumping to countBking
; compNumLoop copies the newly decremented compCount value to its position in the
```

```
    stack and then checks its value against 0. Once it hits 0, the program
; jumps to finished, which allows showComposite to do some stack bookkeeping and
   then return to main
compNumLoop:
mov   [ebp+24], ecx  ; adjusted compCount value copied to its location in stack
cmp   ecx, 0    ; compCount compared to 0
je   finished   ; once 0 is hit, program jumps to finished so that showComposite
   can end
; restart increments mainNum by 1 and then checks to see if it is an even number.
    If it is even, the program automatically jumps to isCompositeCall, so it can
    be printed
; (again, even numbers that aren't 2 are always composite numbers, so there's no
   need to check them). If it is even, it jumps to squareCheck to restart the
   process
; of checking to see if a number is prime or not
restart:
mov   eax, 1    ; incrementing mainNum by 1
add   [ebp+12], eax  ; storing new mainNum value in its location in stack
mov   eax, [ebp+12]  ; moving mainNum back to eax, so that program to check to
   see if it is even or odd
mov   ebx, 2    ; moving 2 to ebx for division
xor   edx, edx   ; clearing edx for division
div   ebx
cmp   edx, 0    ; if remainder is 0, mainNum is odd, so jump to isCompositeCall
mov   eax, [ebp+12]  ; moving mainNum back to eax
je   isCompositeCall
jmp   squareCheck   ; odd numbers have to be checked to see if they are prime or
   composite, so jump to beginning of check process
; finished is a jumping point for compNumLoop. Once compCount hits 0, the program
    has printed the user-specified number of composite numbers. Just need to
; clean up stack a little and return to main
finished:
pop   ebp     ; restoring stack
ret        ; return to main
showComposites ENDP

; isComposite procedure - This procedure prints out composite numbers. It prints
   6 composite numbers per line that are in evenly aligned columns. After
   printing
; 10 lines of composite numbers, it will pause and ask the user to hit enter to
   continue (so, essentially, it is printing 10 lines of composite numbers at a
   time)
; implementation note: isComposite access its paramters via the stack.
; Arguments passed to it must be in this order and in this fashion:
; 1 - printCount value - isComposite is set up to print 6 composite numbers per
   line, so it needs a value to keep track of this. Everytime it prints a
```

```
   composite
; number, it increments printCount by 1. When isComposite ends and returns the
   adjusted printCount value, whatever procedure called it needs to store the
   returned value
; and not alter it (assuming it needs to call isComposite again).
; 2- lineCount value - isComposite is set up to print 10 lines of composite
   numbers. Once it has printed 10, it will pause the program and wait for an
   enter key input
; from the user before it continues. When isComposite ends and returns the
   adjusted lineCount value, whatever procedure called it needs to restore the
   returned value
; and not alter it (assuming it needs to call isComposite again).
; 3 - mainNum value - this is the composite number that isComposite prints.
   isComposite is designed to only be able to properly align numbers that are
   between 1 and 10
; digits long
; receives: printcount value, lineCount value, and mainNum value all on the stack
; returns: adjusted printCount and lineCount values that must not be changed
; preconditions: printCount and lineCount must both be 0 before the first call of
    isComposite. Also, mainNum must be between 1 and 10 digits long for it to be
; properly aligned
; registers changed: eax, ebx, ecx, and edx
isComposite  PROC
push  ebp     ; setting up system stack
mov   ebp, esp
mov   ecx, 9    ; copying 9 to ecx, so that isComposite can generate the
   appropriate number of spaces between each character
mov   ebx, 10    ; copying 10 to ebx, to determine how many digits in mainNum
; fillerDiv checks to see how many digits are in the mainNum value passed to
   isComposite. It divides mainNum by 10. If the result is 0, that means
; mainNum only has 1 digit. The program jumps to printCom where mainCom is
   printed followed by (value in ecx + 5) spaces (in our example, with a 1 digit
; mainNum, the program would print 14 spaces (9 from ecx, since it is was never
   decremented + 5 added on)). If the result isn't 0, ecx is decremented by 1,
; and the value in ebx is multiplied by 10 (we're dividing by powers of 10 (10,
   100, 1000, etc.)), and the process is started all over again. A mainNum with
; 2 digits would get 13 spaces. A mainNum with 10 digits, on the other hand,
   would only get 5 spaces. This all done so that the composite numbers are in
   neatly
; aligned columns. Once fillerDiv determines how many digits are in mainNum, it
   moves to printCom.
fillerDiv:
xor   edx, edx   ; clearing edx for division
div   ebx     ; mainNum by power of 10 to see how many digits it has
cmp   eax, 0    ; when eax equals 0 after division, we have found number of
   digits, so begin printing
```

```
je   printCom
dec   ecx     ; decrementing ecx by 1
cmp   ecx, 0    ; if ecx equals 0, mainNum has 10 digits, so jump to printCom and
    then print minimum 5 spaces
je   printCom
mov   eax, 10    ; getting ready to generate next power of 10
mul   ebx
xchg  ebx, eax   ; swapping next power of 10 back into ebx
mov   eax, [ebp+8]  ; moving mainNum back into eax
jmp   fillerDiv   ; restarting loop
; printCom prints out the composite number and then moves the hex value 20h (
   which equates to a space in the ascii table), so that program can begin
; printing appropriate number of spaces after the number
printCom:
mov   eax, [ebp+8]  ; copying mainNum to eax
call  writeDec   ; printing mainNum
add   ecx, 5    ; adding 5 spaces to number of spaces determined by fillerDiv
mov   al, 20h    ; copying hex value for SPACE on ascii table to al
; printComLoop takes the number of spaces calculated by fillerDiv plus 5 and
   prints THAT many spaces (after the mainNum has been printed by printCom).
   printComLoop
; also increments printCount by 1 and then checks to see if 6 composites have
   been printed on a line. If 6 have been printed, it prints a line break, sets
; printCount to 0, and then increments lineCount by 1. It then checks lineCount
   against 10, and if they are equal, it pauses the program and tells the user
; to hit enter to continue (and waits for that input). When the user hits enter,
   the program unpauses, and printComLoop sets lineCount to 0. The procedure
; then ends
printComLoop:
call  writeChar   ; printing a space
loop  printComLoop  ; printComLoop loops until appropriate amount of spaces have
   been printed
mov   edx, 1    ; once the loop is finished, printCount is incremented by 1
add   [ebp+16], edx
mov   ebx, [ebp+16]  ; new printCount copied to ebx, so it can be compared to 6
cmp   ebx, 6
jne   printDone   ; if not equal to 6, jump to printDone and end procedure
call  crlf    ; if equal to 6, print a line break and set printCount to 0
mov   ebx, 0
mov   [ebp+16], ebx  ; new printCount being copied to position in stack
add   [ebp+12], edx  ; incrementing lineCount by 1
mov   ebx, [ebp+12]  ; copying new lineCount value to ebx, so it can be compared
   to 10
cmp   ebx, 10
jne   printDone   ; if not equal to 10, jump to printDone and end procedure
call  waitmsg    ; if equal to 10, pause program and wait for user to hit enter
```

```
   key
mov   ebx, 0    ; once user hits enter key, set lineCount to 0
mov   [ebp+12], ebx  ; copying lineCount value to position in stack
call  crlf    ; printing line break (after pause) for readability
; printDone is the jumping point to the end of the procedure. it restores the
   stack and exits the procedure
printDone:
pop   ebp     ; restoring stack
ret
isComposite ENDP

; farewell procedure - This procedure takes a passed in string address and prints
    a farewell message to the user
; receives: farewell string address (needs this so it can use writestring to
   print)
; returns: nothing
; preconditions: nothing
farewell proc

push   ebp     ; setting up stack frame
mov    ebp, esp
mov    edx, [ebp+8]  ; copying address of string from stack to edx
call   writestring   ; printing farewell message
pop    ebp     ; restoring stack
ret
farewell ENDP
END main
```