Project 3: RPN Calculator

- **What to do:**

  Ask the user to input a string of expression in RPN form (+ - * /), use a stack to evaluate the result and display the result (in decimal) on screen. The numbers and operations in the RPN expression are assumed to be separated by (one or more) spaces. There will not be negative numbers in the input.

- **Levels of completion:**

  1 A passing grade (80/100): calculate single-digit RPN in valid forms. For example:

  ```
  3  4  -  2  7  +  *  4  /  =  -2 is a valid RPN
  ```

  2 Full score (100/100): allowing multi-digit numbers, and detecting illegal RPN expressions. For example:

  ```
  3  100  105  -  +  433  *  = -866 is a valid RPN
  3  100  105  -  +  *  433  is invalid
  3  abc  105  -  +  433  *  is invalid
  ```

  3 Extra credit (120/100): achieving the above levels for both RPN and PN. For example:

  ```
  3  100  105  -  +  433  *  = -866 is a valid RPN
  -  120  /  *  3  12  - 42  36   = 114 is a valid PN
  -  120  3  100  105  -  +  433  *  is invalid
  ```

- **How to do RPN calculation using stack**:

  a) From left to right, go through every character of the expression, do the following:

  i. if the character is a digit (0 – 9):
  push that number onto stack (hint: you'll need to translate ASCII to number)

  ii. if the character is an op (+ - * /)

  1. pop from the stack a number -> A
  2. pop from the stack a number -> B
  3. calculate B op A (for /, use quotient as result and ignore remainder).
  4. push the result number back onto stack

  b) Pop out from stack -> final result

- **What to turn in (in a zipped file):**
  1. Your assembly file, well organized and commented.
  2. Lab report, containing the following parts:
     a) Program features:
        i. What level does your program achieve? Does your program function correctly on all the features? If not, describe the malfunction in details.
        ii. Have you run into any significant problems / bugs in this project? Describe how they are solved.
     b) I/O demonstrations:
        Provide 5 screenshots on the running I/O of your program to show the behavior of your program. Be thorough cover all the possible cases.
     c) Implementation details:
        i. How did you define the input buffer to get the expression from the user? What is the longest expression you can take? What would happen if the user input something longer than that?
        ii. Explain your stack implementation (data structure, size, stack pointer, operation of push / pop, etc). Does your stack support the checking for overflow / underflow? Provide an example expression that would result in stack overflow, and another example for stack underflow.
        iii. How did you implement the ASCII to number conversion?
        iv. If you implemented level 2, describe in detail how errors are detected. If you implemented level 3, describe (in addition to the error detection mechanism) how you differentiate RPN vs. PN, and how to evaluate a PN expression.
        v. Briefly describe: how would you change your code if the new requirement asks for the support of real numbers (such as 132.633)?
- **You will be graded on the following criteria:**
  1. program functionality
  2. thoroughness and insights in answering the questions in the lab report
  3. code quality
- **Special notice:** same as project 2: If your code looks "similar enough" to somebody else's, both will be deducted 30 points for "lack of originality".
- **Submission notes:** same as project 2.

Kai Zhao 670720413

ECE 267 Spring 2012

2012 May 3

Project 3: RPN Calculator

2.

a) Program features:

i) What level does your program achieve? Does your program function correctly on all the features? If not, describe the malfunction in details.

My program achieved beyond level 3. No, my program does not function correctly on all features. My program is unable to divide by zero to get infinity. Other than that, the features that are NOT implemented in my program are accepting parentheses, evaluating the operator precedence with infix notation, and accepting decimal numbers.

ii) Have you run into any significant problems/bugs in this project? Describe how they were solved.

Yes, the problem I encountered with this project is that this project is just a combination of project 1 (converting ASCII to integer) and project 2 (implementing a stack). Therefore, the problem with this project is that there were no problems with this project. Therefore, neither of the two problems was solved. (The first problem was that a second problem did not exist and the second problem was that it did not exist.)

iii) What have you implemented – basic function, extra credit, anything else?

The user may input any string that is less than 512 characters.

There is enough memory allotted for the stack so that overflow is not possible.

The program will show a warning if the program thinks that the result may be wrong due to an input of a large integer.

Multiple characters does the same operation so that the user will not have to remember which key will end the program because 'q' for quit or 't' for terminate will both work. Also, multiple characters for the same operation will decrease the probability of an invalid character and increase the probability of a valid expression.

Unlike lawyers, this program is case insensitive! Therefore, 'q', 't', 'Q', and 'T' all functions to end the program. Furthermore, any combination of cases of ‘ans’ will refer to the previous answer, which is also implemented.

Adding extra spaces anywhere in any notation is fine.

This program will tell you the notation that it used to compute the answer. It is possible to mix RPN with infix notation and this program will tell you if it used one, the other, or both.

The mod operation is implemented to take the remainder from a division. E.g. a % b.

The power operation is implemented to compute exponents. E.g. a ^ b.

The test operations are implemented to help compute Boolean algebra.

The equal operation allows the user to check if two operands are equal. E.g. a = b.

The less than operation allows to user to check if the first operand is less than the second operand. E.g. a < b.

The greater than operation allows the user to check if the first operand is greater than the second operand. E.g. a > b.

The complement operation allows the user to check if the operand is zero. E.g. a ', a c, or a C.

The negate operation allows the user to negate or invert an integer so that the user does not have to subtract from 0. This operation serves as an alternative to inputting negative integers. E.g. a n, a N, a i, or a I.

The summation operation allows the user to find the sum from 0 to the operand. E.g. a s, or a S.

The factorial operation allows the user to find the factorial of the operand, which is the product from 1 to the operand. E.g. a !.

Although a regular calculator is an embedded system that does allow you the exit the calculator function, the quit operation is implemented to quit or terminate this program for the user's convenience with MARS. E.g. q, Q, t, or T.

The comment operation is implemented. The user my comment on the input to help others understand what the user is trying to do. This is helpful for the 5 screenshots the program assignment asks for. E.g. #.

Negative integers are allowed in any notation. The user may input negative numbers as opposed to having to subtract from 0. E.g. -a

The 'ans' operation in implemented! This operation allows the user to compute calculations using the previous calculated answer. This operation serves as a substitute to using parentheses. Rather than using parentheses, the user may compute the expression in the inner most parenthesis and then just use 'ans' to refer to the previous answer. E.g. ans.

PN notation is implemented! The operators are written before their operands. In PN, the stack is built with values from right to left while operators are evaluated left to right. Therefore, if values themselves involve computations, then the order in which the operators can be evaluated is limited.

Infix notation is implemented! In infix notation, the operators are written in-between their operands. Infix notation allows the user to use this program just like a calculator. Infix notation is the usual way we write expressions. Infix notation needs extra information to make the order of evaluation of the operators clear. However, the standard infix notation

rules were not implemented, so infix notation in this program is always evaluated from left to right regardless of parentheses or operator precedence.

In an attempt to package the [(bugs) and (features that were not implemented)] into a new feature, there is a secret message hidden in this program! In order to unlock and view the secret message, the user has to commit all 10 different bugs or invalid operations in this this program. For user friendliness, toggling a secret code toggle twice will NOT toggle the toggle off. Except for the Intel Pentium bug of "4195835 / 3145727", the user will know when a bug or invalid operation was committed according to the error message.

Note: The 10 secret code toggles corresponds to 1) invalid characters, 2) underflow, 3) dividing by zero, 4) more than 1 item remaining in stack after calculations, 5) negative powers/exponents, 6) negative factorials, 7) use of floating point numbers, 8) use of parentheses, 9) length of inputs above 511 characters, and 10) the Intel Pentium bug of 4195835 / 3145727.

Note: For fun, feel free to test my cryptography to display the secret message without committing all 10 bugs. There are many ways in which the user may modify this program to display the secret message. If you see the message, "That is not going to work.", then the program have caught your trying to modify the code and will reset all the secret code toggles. The user may also scroll to the bottom of the program and follow all the jumps to collect the order of the prompts, and follow all the prompts to get the message, but I imagine that that will be confusing to follow because the callee labels do not follow the prompt labels.

b) Provide 5 screenshots on the running I/O of your program to show how your program behave and illustrate your features. Be thorough to cover all possible cases.

Screenshot 1: Plain RPN and PN

```
3 4 - 2 7 + * 4 /
 = -2 is a valid RPN
34-27+*4/ # forgetting the spaces will result in an error
 is invalid due to underflow
3 100 105 - + 433 *
 = -866 is a valid RPN
- 120 / * 3 12 - 42 36
 = 114 is a valid PN
1 9 - 5 + 4 * # negative answers are possible
 = -12 is a valid RPN
1 2 + 3 + 4 *         #
 = 24 is a valid RPN
# adding extra spaces after the input is fine
1 2 +  3 + # adding in more than 1 space will not result in an error
 = 6 is a valid RPN
```

Screenshot 2: Infix Notation, RPN, Negative Numbers, and Large Numbers

```
1 2 / 3 4 - 5 * +
 = -5 is a valid RPN
1 2 ^ 3 4 - 5 * +
 = -4 is a valid RPN
1 2 ^ / 3 4 5 * +
 = 20 is a valid RPN and infix notation
1 2 ^ / 3 4 - 5 +
 = 1 is a valid RPN and infix notation
1 2 ^ / 3 4 - 5 *
 = -20 is a valid RPN and infix notation
1 + 2 + 3 * 4
 = 24 is a valid infix notation
* 4 + 3 + 1 2
 = 24 is a valid PN
11 + 222 * 3333 # there is no operator precedence
 = 776589 is a valid infix notation
-10 -10 -10 + + # negative RPN
 = -30 is a valid RPN
-1 + 2 -15 + # negative infix and RPN
 = -14 is a valid RPN and infix notation
     -20     +    -3 #negative infix with spaces
 = -23 is a valid infix notation
+ -5 + -49 6 #subtraction in PN
 = -48 is a valid PN
999999999
 = 999999999 is a valid RPN
999999999999
 = -727379969 is a valid RPN
WARNING: The previous result is not to be trusted due to the input of a large integer.
9999999 + 9999999
 = 19999998 is a valid infix notation
999999999999 + 99999999
 = -627379970 is a valid infix notation
WARNING: The previous result is not to be trusted due to the input of a large integer.
```

Screenshot 3: Extra Features and the ‘ans’ function

```
# the comment feature works
1 + 2 - 5 # infix notation
 = -2 is a valid infix notation
ans + 9 # 'ans' works
 = 7 is a valid infix notation
AnS + 20 # any upper/lower case of ans works
 = 27 is a valid infix notation
9 ans + ans + # ans works in RPN
 = 63 is a valid RPN
- aNs 60 # ans works in PN as well
 = 3 is a valid PN
ans ^ ans ^ ans #3^3^3=27^3
 = 19683 is a valid infix notation
ans n
 = -19683 is a valid RPN
ans '
 = 0 is a valid RPN
27 % 7
 = 6 is a valid infix notation
27 % 7 2 +
 = 8 is a valid RPN and infix notation
27 % 7 2 + !
 = 40320 is a valid RPN and infix notation
8 !
 = 40320 is a valid RPN
7 ! * 8
 = 40320 is a valid infix notation
8 s
 = 36 is a valid RPN
s 8
 = 36 is a valid PN
```

Screenshot 4: More Extra Features

```
0 = 0
 = 1 is a valid infix notation
0 = 1
 = 0 is a valid infix notation
0 1 <
 = 1 is a valid RPN
> 0 1
 = 0 is a valid PN
1 s s s s !
 = 1 is a valid RPN
2 s s s
 = 21 is a valid RPN
21 + 52 s + 136 + 23 + 32 s - 136 - 262 - 363 - 4 - 35 63 -
 = 4182415 is a valid RPN and infix notation
  s    6
 = 21 is a valid PN
4 ! s
 = 300 is a valid RPN
4 s ! < ans
 = 0 is a valid infix notation
6 I + 9
 = 3 is a valid infix notation
5 & aNs
 = 15 is a valid infix notation
0 c
 = 1 is a valid RPN
         3      +     2   2  +# extra spaces is fine
 = 7 is a valid RPN and infix notation
           + 4       -    2    3  #extra spaces in fine in any notation
 = 3 is a valid PN
q

-- program is finished running --
```

Screenshot 5: All the Bugs and Errors (How to Unlock the Secret Message)

```
Reset: reset completed.

# the following demonstration will demonstrate almost all the errors,
# bugs, and features within the bugs
qwer # using chars that are not operators nor 0 through 9  cause the char error
 is invalid due to character usage
1 + # not enough operands will cause the underflow error
 is invalid due to underflow
9 / 0 # dividing by zero will cause the division error
 is invalid due to dividing by zero
1 2 # not enough operators will cause the stack height error
 is invalid because there is more than one item remaining in the stack
1 0 9 - ^ # negative exponents will cause the exponent error
 is invalid due to negative exponential
9 i ! # negative factorial will cause the factorial error
 is invalid due to negative factorial
0.1 # '.' will cause the floating point error
 is invalid due to floating point numbers not being implemented
( 5 + 2 ) # '(', or ')' will cause the parentheses error
 is invalid due to parenthesis not being implemented
just type greater than 511 characters for the length bug just type greater than
 is invalid due to length of user input
just type greater than 511 characters for the length bug just type greater than
 is invalid due to length of user input
toggleing the bug again will not toggle the secret message toggles off toggleing
 is invalid due to length of user input
# but the toggles do reset after revealing the message
# try these bugs for youself; the last 1 is the intel bug
4195835 / 3145727 # The message should be revealed after this input
```

c) Implementation details:

i) How did you define the input buffer to get the expression from the user? What is the longest expression you can take? What would happen if the user input something longer than that?

I defined the input buffer as “userInput: .space 508”, followed by “endOfUserInput: .word 0x000a0000”. The longest expression my program can take is 511 characters. If the user input is something longer than 511 characters, then the program will always give a length

error and reset. Furthermore, my program checks 511th character to make sure it is '\n', or else the program will also give a length error and reset because my program does not cut off the user's input before the user is done.

ii) Explain your stack implementation (data structure, size, stack pointer, operation of push / pop, etc.). Does your stack support the checking for overflow / underflow? Provide an example expression that would result in stack overflow, and another example for stack underflow.

The stack is built inverse (later items take on earlier memory address) because of historical precedence as mentioned on page 114 of the book. The stack is 512 bytes (4 bytes for the first item and 508 bytes for the remaining items) so that overflow is not possible. The stack pointer is a global variable saved in $t3, which resets to the first item upon reset after every calculation. The push operation is done by reading in the bytes between '0' and '9', doing the calculations to convert the byte from ASCII to base 10, then saving the integer into the stack. The stack pointer and stack height is increased when a space occurs unless the space is after the user's intended user input, meaning that extra spaces at the end of the input is fine. The pop operation is done whenever an operator is called. In turn, the operator will use methods to decrease the stack height, adjust the stack point accordingly, and load the stack item(s) into register(s). No, my program does not check for overflow because overflow is not possible. Yes, my program does check for underflow by checking if the address pointer is beyond the first item of the stack. If so, my program will convert to infix notation to attempt to salvage the user's input and grab the operand that is after the operator. An example expression that will result in stack overflow is not possible for my program. An example expression that will result in stack

underflow is “1 2 ^ / 3 4 - 5 * +”, and removing any operator will eliminate the underflow.

iii) How did you implement the ASCII to number conversion?

I first subtracted 48 from the ASCII to convert the ASCII to an integer. By subtracting 48, I am essentially subtracting 0x30, causing 0x3n → n, where n is an inclusive integer from 0 through 9. If the calculator is in RPN or infix notation mode, then the current push value is multiplied by 10, the base, and summed with the current integer value. If the calculator is in PN, then the current integer is multiplied by a power of 10, and summed with the current integer value. Register $t5 is the power of 10 that resets to 1 after every push, and $t5 also increases by a factor of 10 after every new byte in PN so that the program is adding and pushing the correct values.

iv) If you implemented level 2, describe in detail how errors are detected. If you implemented level 3, describe (in addition to the error detection mechanism) how you differentiate RPN vs. PN, and how to evaluate a PN expression.

The invalid character usage is detected by checking if the user attempts to use any character that is not [(an operator) or (between 0 through 9)]. Underflow detection is done by checking if the stack pointer is beyond the first item. The dividing by zero error is detected by checking if the second operand is 0 in the division procedure. Having more than 1 item remaining in the stack after the calculation is detected by checking the stack height right before displaying the answer. The negative exponent error is detected by checking if the second operand is less than zero before computing the exponent. The negative factorial is detected by checking if the second operand is less than zero before computing the factorial. The floating point error and parentheses error is detected if the user tries to use

'.', '(', or ')'. The length error is detect by checking if the 511th character is '/n' to make sure the user has press enter as opposed to cutting the user off. The program will go into PN mode if the first character is not an integer. The program will go into infix notation mode if an underflow is about to occur and if switching to infix notation will prevent the underflow error. RPN, PN, and infix notation each had their own register as a toggle so that the program can decide which mode the program should use. PN expression are evaluated the same way in the sense that the operator did not change, only the method that the operators used to place the operands into position changed as described earlier in the ASCII to number conversion.

v) Briefly describe: how would you change your code if the new requirement asks for the support of real numbers (such as 132.633)?

If the requirement asks for support of real numbers, then I will switch over to floating point notation. Each item in the stack will be allowed to be 8 bytes long, 4 bytes for the integer values and 4 bytes for the decimal values. The program will read ASCII values into the register for integers until the decimal point is detect, which then the program will read ASCII values into the register for decimal values. Next, the program will convert from ASCII to integer values since to the way my program currently does and from integer to binary values and hex values. E.g. integer register 0x323331 → $(132)_{10}$ → $(1000\ 0100)_2$ → 0x00000084. The program then will use the decimal value to binary value conversion algorithm (multiply by 2, subtract 1 if the product is ≥ 1, and repeat) to convert the decimal value register. E.g. decimal value register 0x333336 → $(.633)_{10}$ → $(1010\ 0010\ 0000\ 1100\ \ldots)_2$ → 0xa20c…. The program will then shift logic right from the register with the integer value into the register with the binary point values until there is

0x00000001 remaining in the integer value while using another register to count the number of shifts. If the program has 0 as the integer value, then logic will be shifted left until there is 0x00000001 remaining in the integer value while using another register to count the number of shifts. E.g. 132 = $(1000\ 0100)_2$ will require 7 shifts to the right to get $(0000\ 0001)_2$. 132.633 → 0000 1001 0100 0100 0001 1000 1001 0011→ 0x09441893. Since only the first 23 bits of the register that carries the binary point value can be used as the mantissa in single precision, the logic will continue to be shifted to the right 9 times so that the first 9 bits are all zeros. E.g. 0x09441893 → 0x0004a20c. Next, 127 will be added into the register that counted the number of shifts to get the correct value of E because E is biased in floating point notation. E.g. for 132, 7 shifts were required to get the register with the integer value from $(1000\ 0100)_2$ to $(1)_2$. Adding 127 to 7 is 134, which is $(1000\ 0110)_2$ → 0x00000086. Logic will then be shifted left 23 times to get E in the correct position. E.g. the register with E goes from 0x00000086 → 0x43000000. The register containing the E will be summed with the register containing the mantissa. E.g. 0x43000000 + 0x0004a20c = 0x4304a20c, which is equal to 132.633. The floating point representation of numbers will then be pushed into the stack. E.g. push in 0x4304a20c because it represents 132.633. Other values will be read, converted, and pushed into the stack in a similar manner.

Operands will pop the top 2 items of the stack and placed into $f1 and $f2 register. The operands will then use the FP commands to compute the operation, and the result will be pushed back into the stack. At the end of the user input, 2 will be loaded into $v0 to print the floating point number representation of the solution using a syscall.