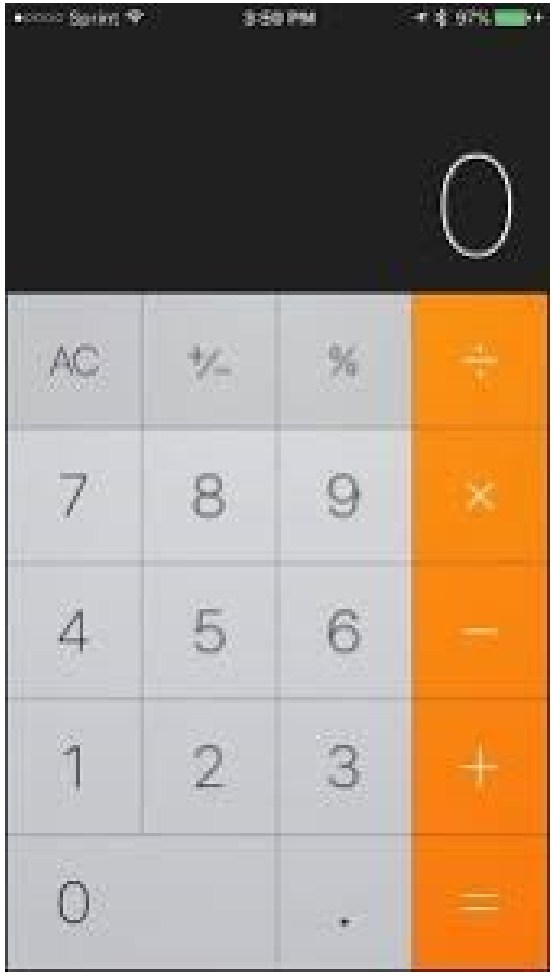
ECE 220 Computer Systems & Programming

Lecture 5 – Programming with Stack

January 29, 2019

ECE ILLINOIS

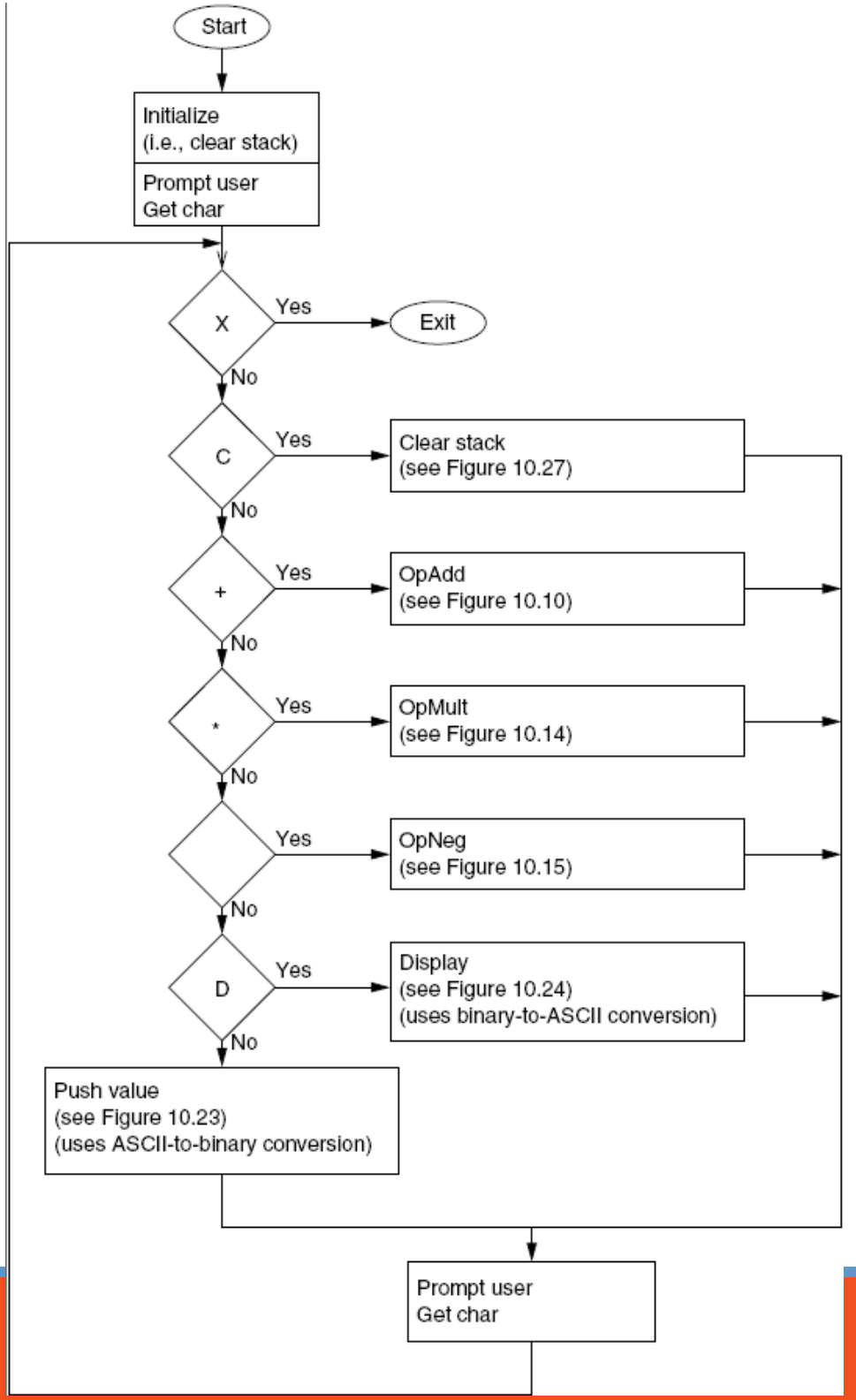
ILLINOIS



X- Exit the Simulation
D- Display the Result from Stack Top
C-Clear Stack
+ Perform Addition
- Negate the top element on the stack
***** Perform Multiplication
Enter – Push the typed data onto the stack

Flow Chart of the Calculator

X- Exit the Simulation
D- Display the Result from Stack Top
C-Clear Stack
+ Perform Addition
- Negate the top element on the stack
***** Perform Multiplication
Enter – Push the typed data onto the stack



What are the different subroutines we need?

- OpAdd (Adder subroutine)
- OpMult (Multiplication subroutine)
- OpNeg (Negate subroutine)
- Range Check (integers in the range -999 and +999)
- AsciitoBinary (input data type conversion)
- BinarytoAscii (output data type conversion)
- PushValue subroutine (which push input onto the Stack)
- OpDisplay (Which pop the result from stack and display Ascii result on the screen)
- OpClear (which clear the stack)

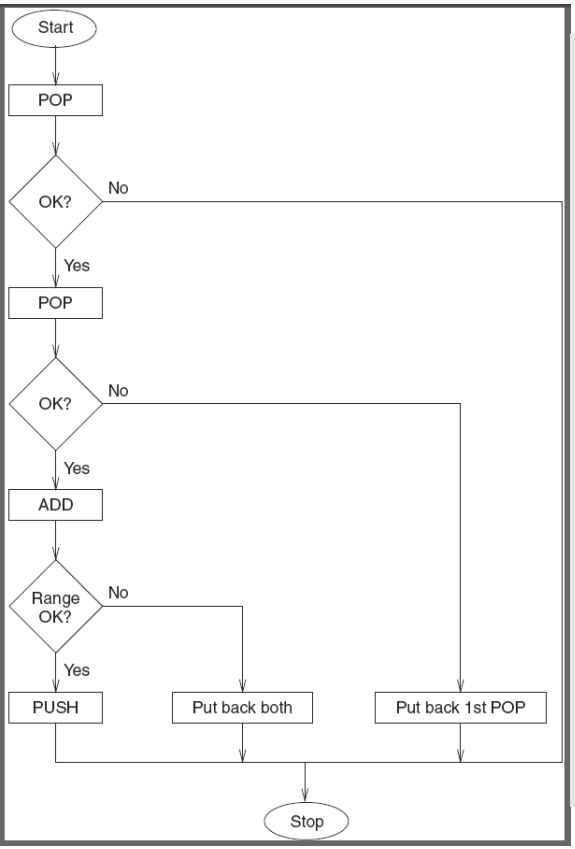
Main Function:

```
;
;  The Calculator, Main Algorithm
;
.ORIG   x3000
                LEA         R6,StackBase  ; Initialize the Stack.
                ADD         R6,R6,#1      ; R6 is stack pointer
                LEA         R0,PromptMsg
                PUTS
                GETC
                OUT
;
; Check the command
;
Test            LD          R1,NegX       ; Check for X
                ADD         R1,R1,R0
                BRz         Exit
;
                LD          R1,NegC       ; Check for C
                ADD         R1,R1,R0
                BRz         OpClearC
;
                LD          R1,NegPlus    ; Check for +
                ADD         R1,R1,R0
                BRz         OpAddC
;
                LD          R1,NegMult    ; Check for *
                ADD         R1,R1,R0
                BRz         OpMultC
;
                LD          R1,NegMinus   ; Check f
                ADD         R1,R1,R0
                BRz         OpNegC
;
                LD          R1,NegD       ; Check f
                ADD         R1,R1,R0
                BRz         OpDisplayC
;
; Then we must be entering an integer
;
                JSR     PushValue     ; See Figure 10.23
```

```
OpClearC    JSR OpClear
OpAddC      JSR OpAdd
OpMultC     JSR OpMult
OpNegC      JSR OpNeg
OpDisplayC  JSR OpDisplay

Exit                HALT
PromptMsg           .FILL     x000A
                    .STRINGZ "Enter a command:"
NegX                .FILL     xFFA8
NegC                .FILL     xFFBD
NegPlus             .FILL     xFFD5
NegMinus            .FILL     xFFD3
NegMult             .FILL     xFFD6
NegD                .FILL     xFFBC
```

OpAdd

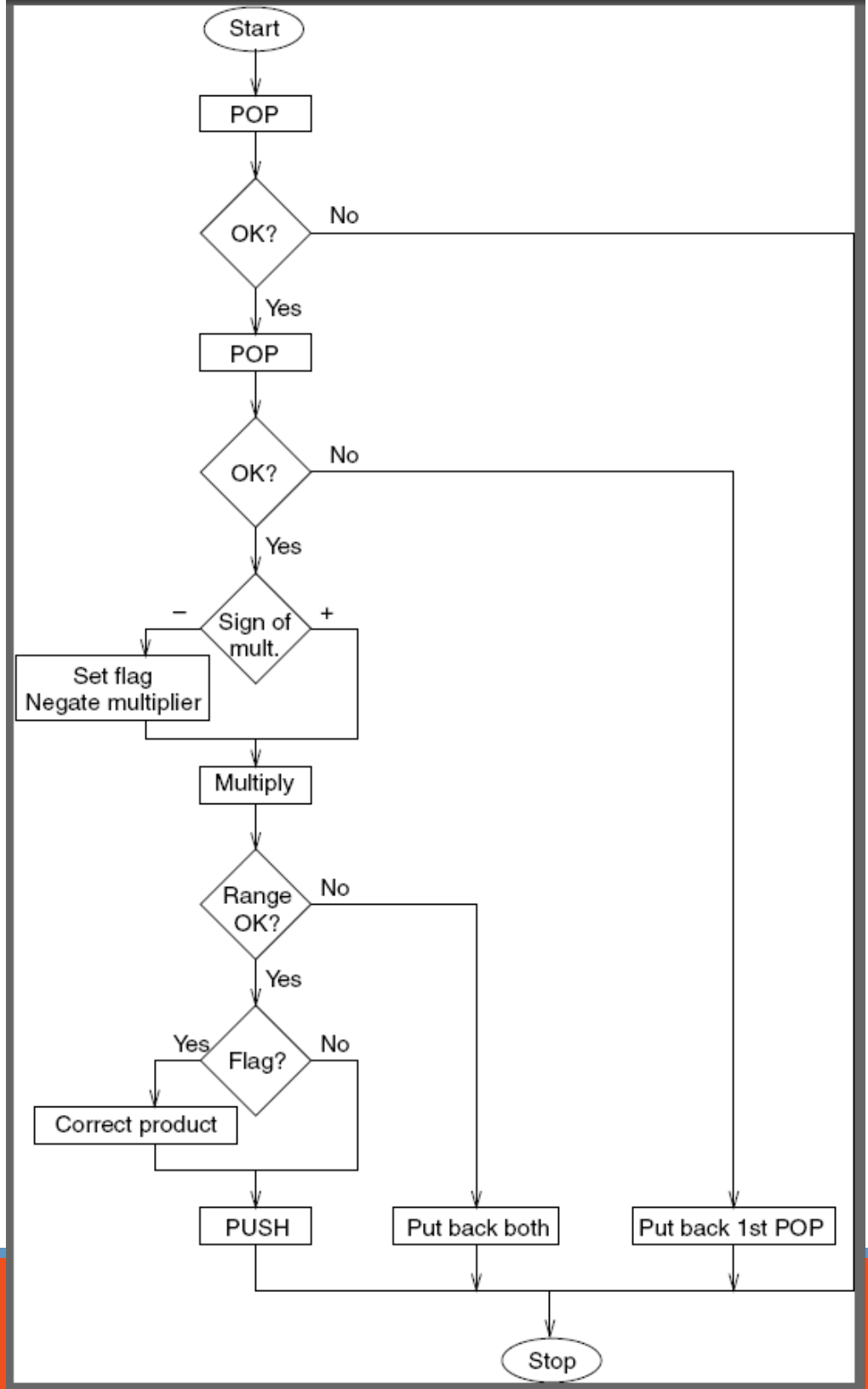



```
;         Routine to pop the top two elements from the stack,
;         add them, and push the sum onto the stack.  R6 is
;         the stack pointer.
;
OpAdd       JSR     POP             ; Get first source operand.
            ADD     R5,R5,#0        ; Test if POP was successful.
            BRp     Exit_A            ; Branch if not successful.
            ADD     R1,R0,#0        ; Make room for second operand
            JSR     POP             ; Get second source operand.
            ADD     R5,R5,#0        ; Test if POP was successful.
            BRp     Restore1_A        ; Not successful, put back first.
            ADD     R0,R0,R1        ; THE Add.
            JSR     RangeCheck      ; Check size of result.
            BRp     Restore2_A        ; Out of range, restore both.
            JSR     PUSH            ; Push sum on the stack.
            JSR  NewCommand       ; Return to the Main Program
Restore2_A    ADD     R6,R6,#-1       ; Decrement stack pointer.
Restore1_A    ADD     R6,R6,#-1       ; Decrement stack pointer.
Exit_A        JSR  NewCommand       ; Return to the Main Program
```

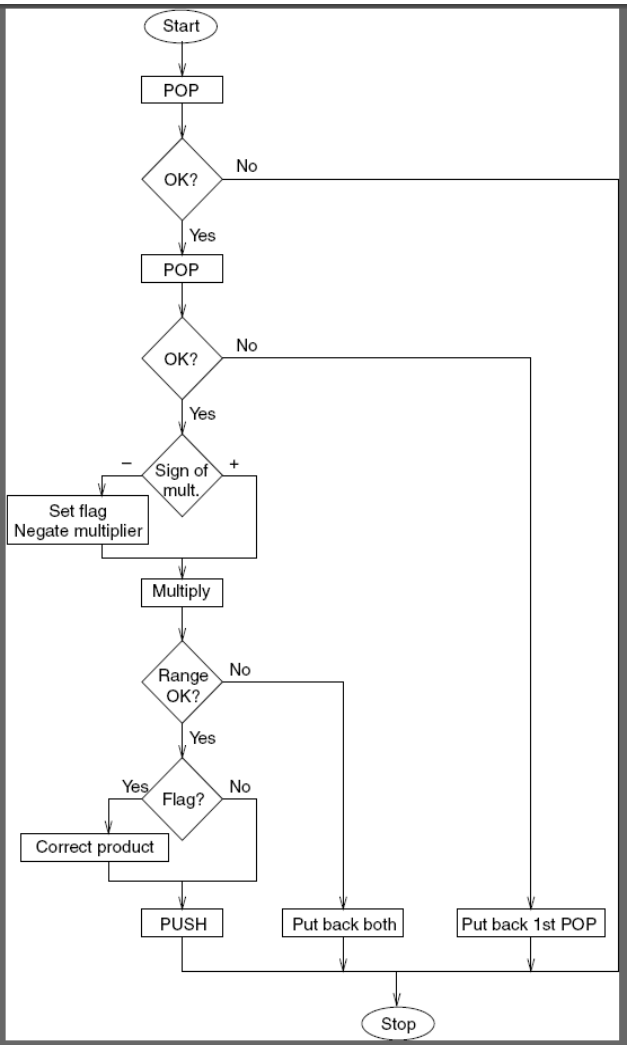
OpNeg Subroutine

```
;    Algorithm to pop the top of the stack, form its negative,
;    and push the result on the stack.
;
OpNeg           JSR     POP             ; Get the source operand
                ADD     R5,R5,#0        ; test for successful pop
                BRp     Exit_N            ; Branch if failure
                NOT     R0,R0
                ADD     R0,R0,#1        ; Form the negative of the source.
                JSR     PUSH            ; Push the result on the stack.
Exit_N          JSR   NewCommand        ; Return to the Main Program
```

OpMult:

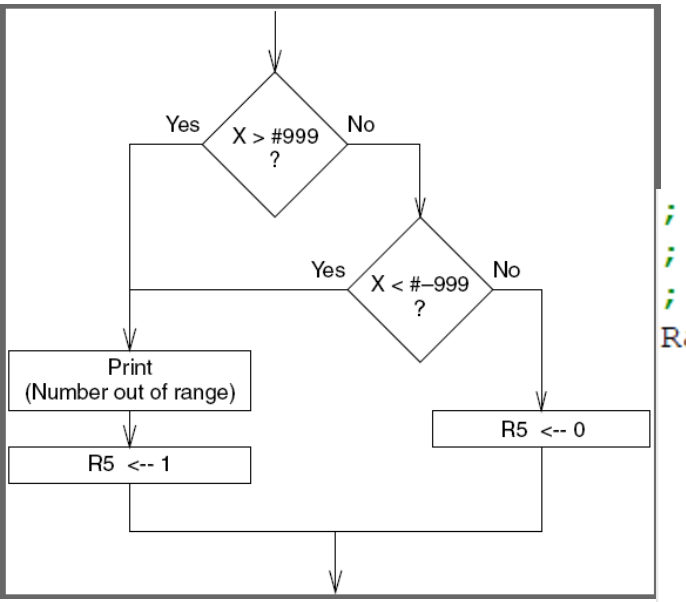

Start
POP
OK?
No
Yes
POP
OK?
No
Yes
Sign of mult.
–
+
Set flag
Negate multiplier
Multiply
Range OK?
No
Yes
Flag?
Yes
No
Correct product
PUSH
Put back both
Put back 1st POP
Stop

OpMult (code)



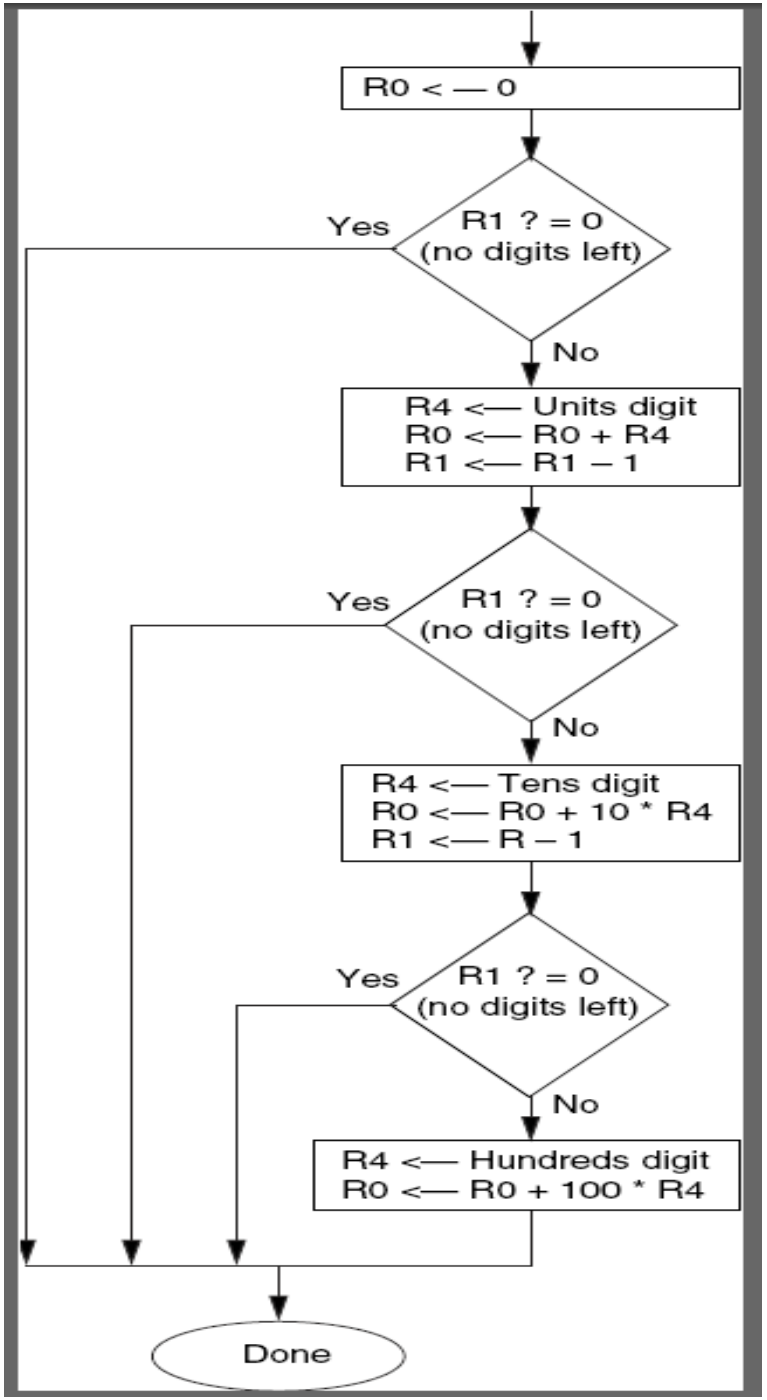

```
;    Algorithm to pop two values from the stack, multiply them
;    and if their product is within the acceptable range, push
;    the result on the stack.  R6 is stack pointer.
;
OpMult          AND     R3,R3,#0        ; R3 holds sign of multiplier.
                JSR     POP             ; Get first source from stack.
                ADD     R5,R5,#0        ; Test for successful POP
                BRp     Exit_M            ; Failure
                ADD     R1,R0,#0        ; Make room for next POP
                JSR     POP             ; Get second source operand
                ADD     R5,R5,#0        ; Test for successful POP
                BRp     Restore1_M        ; Failure; restore first POP
                ADD     R2,R0,#0        ; Moves multiplier, tests sign
                BRzp    PosMultiplier
                ADD     R3,R3,#1        ; Sets FLAG: Multiplier is neg
                NOT     R2,R2
                ADD     R2,R2,#1        ; R2 contains -(multiplier)
PosMultiplier   AND     R0,R0,#0        ; Clear product register
                ADD     R2,R2,#0
                BRz     PushMult        ; Multiplier = 0, Done.
;
MultLoop        ADD     R0,R0,R1        ; THE actual "multiply"
                ADD     R2,R2,#-1       ; Iteration Control
                BRp     MultLoop
;
                JSR     RangeCheck
                ADD     R5,R5,#0        ; R5 contains success/failure
                BRp     Restore2_M
;
                ADD     R3,R3,#0        ; Test for negative multiplier
                BRz     PushMult
                NOT     R0,R0           ; Adjust for
                ADD     R0,R0,#1        ; sign of result
PushMult        JSR     PUSH            ; Push product on the stack.
                JSR  NewCommand     ; Return to the Main Program
Restore2_M      ADD     R6,R6,#-1       ; Adjust stack pointer.
Restore1_M      ADD     R6,R6,#-1       ; Adjust stack pointer.
Exit_M          JSR  NewCommand     ; Return to the Main Program
```

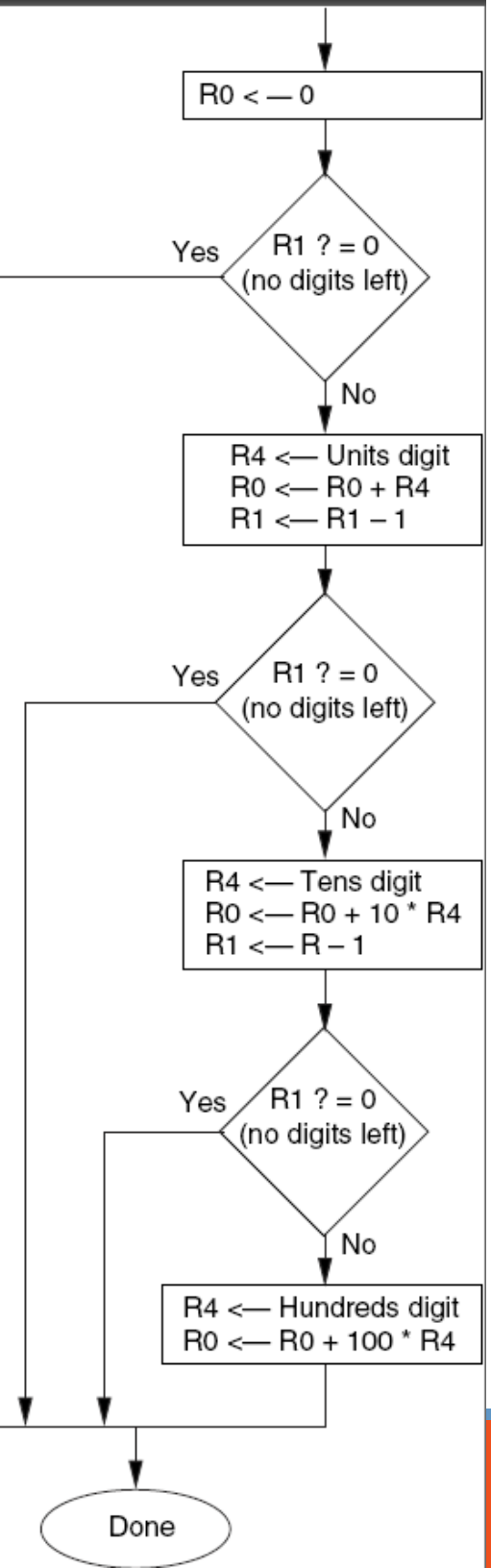
RangeCheck




```
;       Routine to check that the magnitude of a value is
;       between -999 and +999.
;
RangeCheck      LD      R5,Neg999
                ADD     R4,R0,R5    ; Recall that R0 contains the
                BRp     BadRange    ; result being checked.
                LD      R5,Pos999
                ADD     R4,R0,R5
                BRn     BadRange
                AND     R5,R5,#0    ; R5 <-- success
                RET
BadRange        ST      R7,Save_R     ; R7 is needed by TRAP/RET
                LEA     R0,RangeErrorMsg
                TRAP    x22         ; Output character string
                LD      R7,Save_R
                AND     R5,R5,#0    ;
                ADD     R5,R5,#1    ; R5 <-- failure
                RET
Neg999          .FILL   #-999
Pos999          .FILL   #999
Save_R            .FILL     x0000
RangeErrorMsg   .FILL   x000A
                .STRINGZ  "Error: Number is out of range."
```

AsciitoBinary






```
;  This algorithm takes an ASCII string of three decimal digits and
;  converts it into a binary number.  R0 is used to collect the result.
;  R1 keeps track of how many digits are left to process.  ASCIIBUFF
;  contains the most significant digit in the ASCII string.
;
ASCIItoBinary   AND     R0,R0,#0        ; R0 will be used for our result
                ADD     R1,R1,#0        ; Test number of digits.
                BRz     DoneAtoB        ; There are no digits
;
                LD      R3,NegASCIIOffset  ; R3 gets xFFD0, i.e., -x0030
                LEA     R2,ASCIIBUFF
                ADD     R2,R2,R1
                ADD     R2,R2,#-1       ; R2 now points to "ones" digit
;
                LDR     R4,R2,#0        ; R4 <-- "ones" digit
                ADD     R4,R4,R3        ; Strip off the ASCII template
                ADD     R0,R0,R4        ; Add ones contribution
;
                ADD     R1,R1,#-1
                BRz     DoneAtoB        ; The original number had one digit
                ADD     R2,R2,#-1       ; R2  now points to "tens" digit
;
                LDR     R4,R2,#0        ; R4 <-- "tens" digit
                ADD     R4,R4,R3        ; Strip off ASCII  template
                LEA     R5,LookUp10     ; LookUp10 is BASE of tens values
                ADD     R5,R5,R4        ; R5 points to the right tens value
                LDR     R4,R5,#0
                ADD     R0,R0,R4        ; Add tens contribution to total
;
                ADD     R1,R1,#-1
                BRz     DoneAtoB        ; The original number had two digits
                ADD     R2,R2,#-1       ; R2 now points to "hundreds" digit
;
                LDR     R4,R2,#0        ; R4 <-- "hundreds" digit
                ADD     R4,R4,R3        ; Strip off ASCII template
                LEA     R5,LookUp100    ; LookUp100 is hundreds BASE
                ADD     R5,R5,R4        ; R5 points to hundreds value
                LDR     R4,R5,#0
                ADD     R0,R0,R4        ; Add hundreds contribution to total
DoneAtoB        RET
```

AsciitoBinary (Continued)

```
NegASCIIOffset .FILL   xFFD0
ASCIIBUFF      .BLKW   #4
LookUp10       .FILL   #0
               .FILL   #10
               .FILL   #20
               .FILL   #30
               .FILL   #40
               .FILL   #50
               .FILL   #60
               .FILL   #70
               .FILL   #80
               .FILL   #90
;
LookUp100      .FILL   #0
               .FILL   #100
               .FILL   #200
               .FILL   #300
               .FILL   #400
               .FILL   #500
               .FILL   #600
               .FILL   #700
               .FILL   #800
               .FILL   #900
```

BinarytoAscii

```
;  This algorithm takes the 2's complement representation of a signed
;  integer, within the range -999 to +999, and converts it into an ASCII
;  string consisting of a sign digit, followed by three decimal digits.
;  R0 contains the initial value being converted.
;
BinarytoASCII   LEA   R1,ASCIIBUFF  ; R1 points to string being generated
                ADD   R0,R0,#0      ; R0 contains the binary value
                BRn   NegSign       ;
                LD    R2,ASCIIplus  ; First store the ASCII plus sign
                STR   R2,R1,#0
                BRnzp Begin100
NegSign         LD    R2,ASCIIminus ; First store ASCII minus sign
                STR   R2,R1,#0
                NOT   R0,R0         ; Convert the number to absolute
                ADD   R0,R0,#1      ; value; it is easier to work with.
Begin100        LD    R2,ASCIIoffset ; Prepare for "hundreds" digit
                LD    R3,Neg100     ; Determine the hundreds digit
Loop100         ADD   R0,R0,R3
                BRn   End100
                ADD   R2,R2,#1
                BRnzp Loop100
End100          STR    R2,R1,#1    ; Store ASCII code for hundreds digit
                LD     R3,Pos100
                ADD    R0,R0,R3    ; Correct R0 for one-too-many subtracts
;
                LD     R2,ASCIIoffset ; Prepare for "tens" digit
;
Begin10         LD     R3,Neg10    ; Determine the tens digit
Loop10          ADD    R0,R0,R3
                BRn    End10
                ADD    R2,R2,#1
                BRnzp  Loop10
;
End10           STR    R2,R1,#2    ; Store ASCII code for tens digit
                ADD    R0,R0,#10   ; Correct R0 for one-too-many subtracts
Begin1          LD     R2,ASCIIoffset ; Prepare for "ones" digit
                ADD    R2,R2,R0
                STR    R2,R1,#3
                RET
```

```
ASCIIplus       .FILL   x002B
ASCIIminus      .FILL   x002D
ASCIIoffset     .FILL   x0030
Neg100          .FILL   xFF9C
Pos100          .FILL   x0064
Neg10           .FILL   xFFF6
```

ILLINOIS

PushValue:

```
1  ; This algorithm takes a sequence of ASCII digits typed by the user,
2  ; converts it into a binary value by calling the ASCIItoBinary
3  ; subroutine and pushes the binary value onto the stack.
4  ;
5  PushValue       LEA     R1,ASCIIBUFF  ; R1 points to string being
6                  LD      R2,MaxDigits  ; generated
7  ;
8  ValueLoop       ADD     R3,R0,xFFF6   ; Test for carriage return
9                  BRz     GoodInput
10                 ADD     R2,R2,#0
11                 BRz     TooLargeInput
12                 ADD     R2,R2,#-1     ; Still room for more digits
13                 STR     R0,R1,#0      ; Store last character read
14                 ADD     R1,R1,#1
15                 GETC
16                 OUT                   ; Echo it
17                 BRnzp   ValueLoop
18 ;
19 GoodInput       LEA     R2,ASCIIBUFF
20                 NOT     R2,R2
21                 ADD     R2,R2,#1
22                 ADD     R1,R1,R2      ; R1 now contains no. of char.
23                 JSR     ASCIItoBinary
24                 JSR     PUSH
25                 BRnzp   NewCommand
26 ;
27 TooLargeInput   GETC                  ; Spin until carriage return
28                 OUT
29                 ADD     R3,R0,xFFF6
30                 BRnp    TooLargeInput
31                 LEA     R0,TooManyDigits
32                 PUTS
33                 BRnzp   NewCommand
34 TooManyDigits   .FILL    x000A
35                 .STRINGZ "Too many digits"
36 MaxDigits       .FILL    x0003
```

PushValue

```
; This algorithm takes a sequence of ASCII digits typed by the user,
; converts it into a binary value by calling the ASCIItoBinary
; subroutine and pushes the binary value onto the stack.
;
PushValue       LEA     R1,ASCIIBUFF  ; R1 points to string being
                LD      R2,MaxDigits  ; generated
;
ValueLoop       ADD     R3,R0,xFFF6   ; Test for carriage return
                BRz     GoodInput
                ADD     R2,R2,#0
                BRz     TooLargeInput
                ADD     R2,R2,#-1     ; Still room for more digits
                STR     R0,R1,#0      ; Store last character read
                ADD     R1,R1,#1
                GETC
                OUT                   ; Echo it
                BRnzp   ValueLoop
;
GoodInput       LEA     R2,ASCIIBUFF
                NOT     R2,R2
                ADD     R2,R2,#1
                ADD     R1,R1,R2      ; R1 now contains no. of char
                JSR     ASCIItoBinary
                JSR     PUSH
                JSR     NewCommand
;
TooLargeInput   GETC                  ; Spin until carriage return
                OUT
                ADD     R3,R0,xFFF6
                BRnp    TooLargeInput
                LEA     R0,TooManyDigits
                PUTS
                JSR     NewCommand
TooManyDigits   .FILL   x000A
                .STRINGZ "Too many digits"
MaxDigits       .FILL   x0003
```

POP Subroutine

```
;  This algorithm POPs a value from the stack and puts it in
;  R0 before returning to the calling program.  R5 is used to
;  report success (R5=0) or failure (R5=1) of the POP operation.
POP             LEA     R0,StackBase
                NOT     R0,R0
                ADD     R0,R0,#1        ; R0 = -addr.ofStackBase
                ADD     R0,R0,R6        ; R6 = StackPointer
                BRz     Underflow
                LDR     R0,R6,#0        ; The actual POP
                ADD     R6,R6,#1       ; Adjust StackPointer
                AND     R5,R5,#0        ; R5 <-- success
                RET
Underflow       ST      R7,Save_P         ; TRAP/RET needs R7
                LEA     R0,UnderflowMsg
                PUTS                    ; Print error message.
                LD      R7,Save_P         ; Restore R7
                AND     R5,R5,#0
                ADD     R5,R5,#1        ; R5 <-- failure
                RET
Save_P            .FILL   x0000
StackMax        .BLKW   #9
StackBase       .FILL   x0000
UnderflowMsg    .FILL   x000A
                .STRINGZ "Error: Too Few Values on the Stack."
```

PUSH Subroutine

```
;  This algorithm PUSHes on the stack the value stored in R0.
;  R5 is used to report success (R5=0) or failure (R5=1) of
;  the PUSH operation.
;
PUSH            ST      R1,Save1_push       ; R1 is needed by this routine
                LEA     R1,StackMax
                NOT     R1,R1
                ADD     R1,R1,#1        ; R1 = - addr. of StackMax
                ADD     R1,R1,R6        ; R6 = StackPointer
                BRz     Overflow
                ADD     R6,R6,#-1       ; Adjust StackPointer for PUSH
                STR     R0,R6,#0        ; The actual PUSH
                BRnzp   Success_exit
Overflow        ST      R7,Save_push
                LEA     R0,OverflowMsg
                PUTS
                LD      R7,Save_push
                LD      R1, Save1_push      ; Restore R1
                AND     R5,R5,#0
                ADD     R5,R5,#1        ; R5 <-- failure
                RET
Success_exit    LD      R1,Save1_push       ; Restore R1
                AND     R5,R5,#0        ; R5 <-- success
                RET
Save_push           .FILL   x0000
Save1_push          .FILL   x0000
OverflowMsg     .STRINGZ "Error: Stack is Full."
```

OpDisplay and OpClear

```
; This algorithm calls BinarytoASCII to convert the 2's complement
; number on the top of the stack into an ASCII character string, and
; then calls PUTS to display that number on the screen.
OpDisplay    JSR       POP          ; R0 gets the value to be displayed
             ADD       R5,R5,#0
             BRp       NewCommandC  ; POP failed, nothing on the stack.
             JSR       BinarytoASCII
             LD        R0,NewlineChar
             OUT
             LEA       R0,ASCIIBUFF
             PUTS
             ADD       R6,R6,#-1    ; Push displayed number back on stack
             JSR     NewCommand

NewCommandC JSR NewCommand

NewlineChar  .FILL     x000A
;

;
; This routine clears the stack by resetting the stack pointer (R6).
;
OpClear      LEA       R6,StackBase  ; Initialize the Stack.
             ADD       R6,R6,#1      ; R6 is stack pointer
             JSR     NewCommand
```