Because learning changes everything.®

From Bits and Gates to C and Beyond

A Calculator

Chapter 10

Calculator: A Comprehensive Example

Write a program that implements a simple arithmetic calculator.

- Operations: add, subtract, and multiply integers.
- User interface: input characters from keyboard, output to monitor.
- Stack-based calculator -- integers are pushed onto the stack and then operations are performed on the stack values.

Sample Run:

```
Enter a command: 123
Enter a command: 30
Enter a command: +
Enter a command: D
153
```

Push values onto the stack

Pop the top two values, add them, and push the result

Display (print) the value at the top of stack

User input in bold

Program output

Restriction: Integer inputs and results must be with the range −999 to +999.

Program Organization

Main program + 11 subroutines

- ASCII to binary conversion.
- Binary to ASCII conversion.
- Push value (from user) to the stack.
- Display (print) value at top of stack.
- Add.
- Multiply.
- Negate.
- Range check.
- Clear stack.
- Push (from Chapter 8).
- Pop (from Chapter 8).

<u>Topics that need further explanation</u>

Converting between ASCII and 2's complement

Arithmetic using a stack

Data Type Conversion: ASCII Strings to Binary Integers

User input is based on ASCII strings -- characters typed on the keyboard.

In this program, users type decimal integers: "123."
A linefeed (or Enter) must by typed after each input.

LC-3 operations work on 2's complement binary integers, not strings.
So we have to convert the string "123" into the 2's complement representation of the integer 123.

Not all numbers are three digits -- user might type "6" or "30."

Key insight: Each digit corresponds to a multiple of 1, 10, or 100, depending on its position in the string.

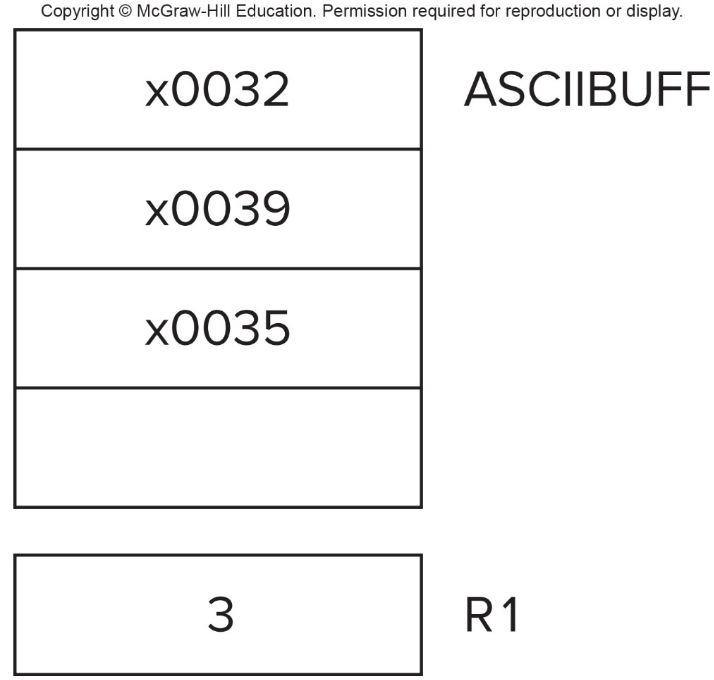
ASCII to Binary: Data Storage

As characters are read, they will be stored in an array called ASCIIBUFF. Instead of using a null terminator for the string, we will track how many digits were entered.

The figure shows "295" in the buffer.
The ASCII codes for '2' and '9' and '5' are in memory, and R1 tells us that three digits were entered.

Since no null terminator, why do we show four memory locations for the buffer? This same buffer will be used for converting binary to ASCII, and we will need space for a minus sign '−' for negative values.

Why not use a null terminator? We are doing character-by-character I/O and we have a limited number of characters -- the null terminator doesn’t provide any benefit.



Access the text alternative for slide images.

ASCII to Binary: Computation

Given the data structure on the previous slide, we can easily identify the ones digit, tens digit (if any), and hundreds digit (if any). Need to do the following:

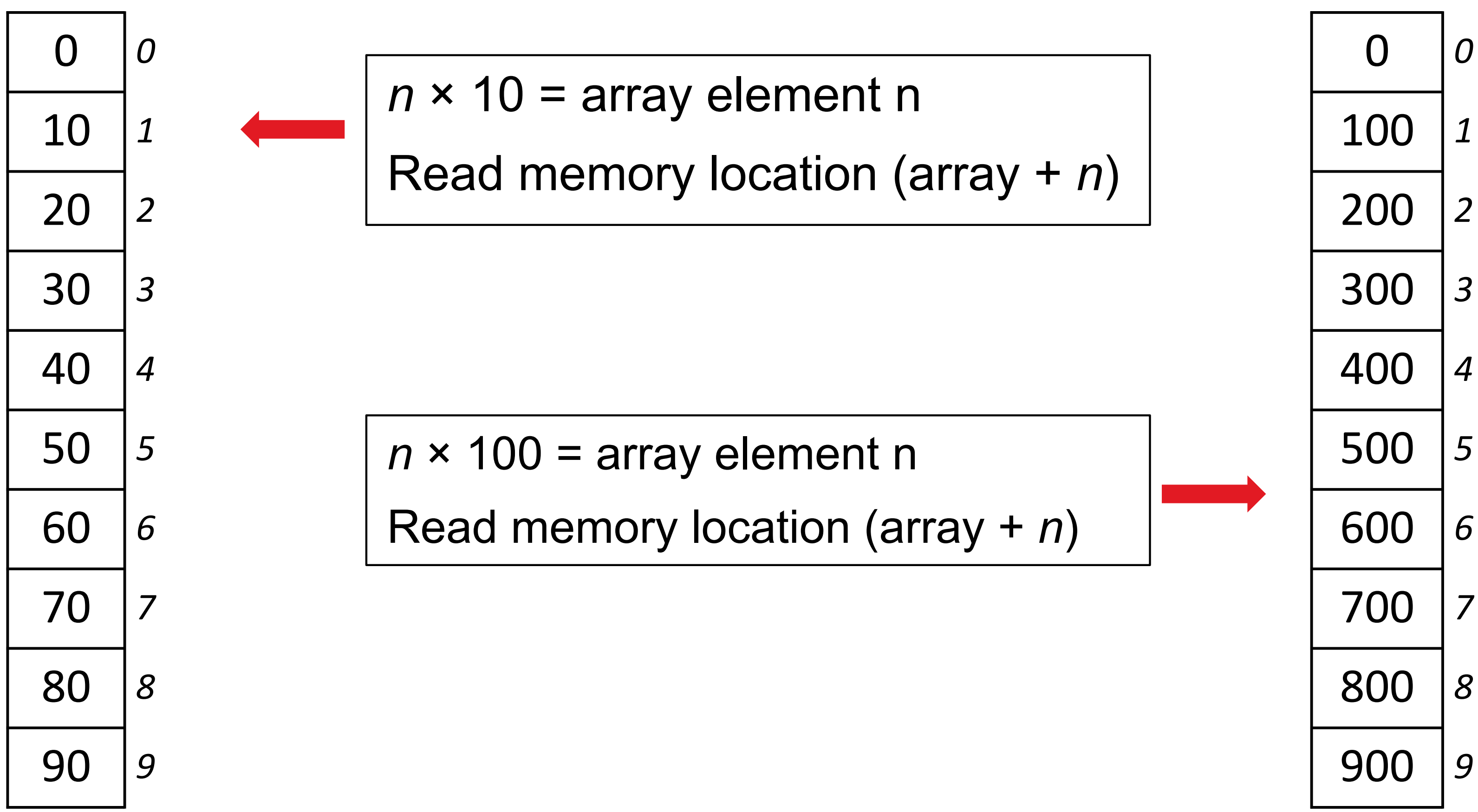
- Convert ones digit to a binary value (subtract x30).
- Convert tens digit to binary value and multiply by 10.
- Convert hundreds digit to binary value and multiply by 100.
- Add the three values together.

No multiply instruction? Next slide...

ASCII to Binary: Multiply by Lookup Table

We do not need a general multiply routine, because there are only 10 interesting multiples of ten and 10 interesting multiples of 100.

Use a lookup table -- put the multiples of 10 (or 100) in an array, and choose the right value based on an index.



ASCII to Binary: Algorithm

Initialize R0 (result) to 0.
R1 is number of digits (1, 2, or 3).
Initialize pointer (R2) to ASCIIBUFF + R1 - 1.

If R1 > 0:

R4 = M[R2] (ones digit)

Convert R4 to binary value (0-9).

Add to R0.

Decrement R1.

If R1 > 0:

R4 = M[R2-1] (tens digit)

Convert R4 to binary value (0-9).

Multiply by 10 and add to R0.

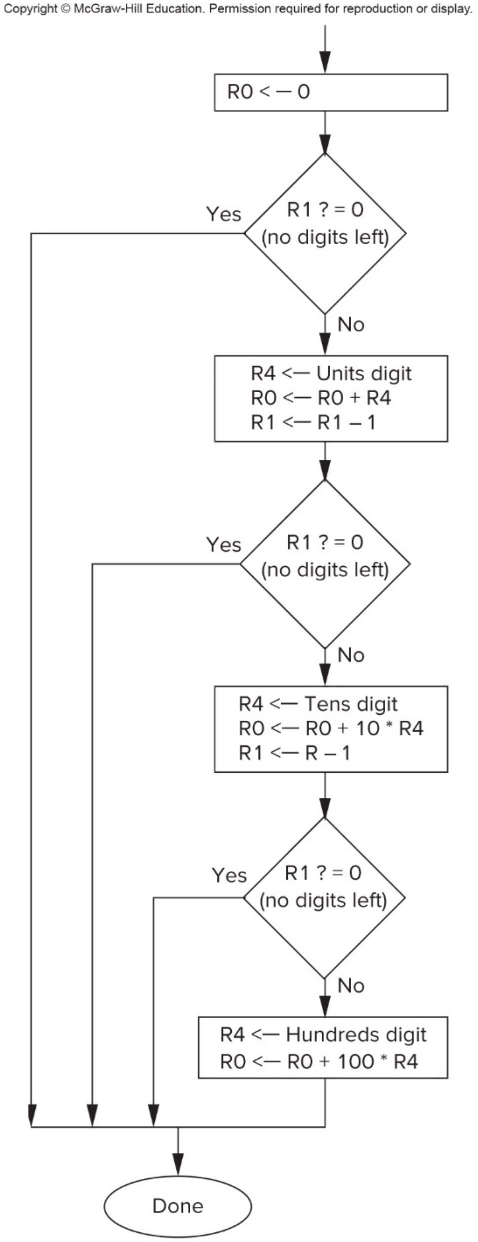
Decrement R1.

If R1 > 0:

R4 = M[R2-2] (hundreds digit)

Convert R4 to binary value (0-9).

Multiply by 100 and add to R0.



Access the text alternative for slide images.

ASCII to Binary Subroutine: Part 1

```
; Convert ASCII string to 2's complement binary.
; ASCIIBUFF is address of string buffer. R1 is number of digits.
; Result returned in R0.

ASCIItoBinary     ST    R1, AtoB_SR1    ; save registers
                  ST    R2, AtoB_SR2
                  ST    R3, AtoB_SR3
                  ST    R4, AtoB_SR4
                  AND   R0, R0, #0      ; initialize result
                  ADD   R1, R1, #0      ; if zero digits, result is 0
                  BRz   AtoB_DONE

                  LD    R2, AtoB_ASCIIBUFF
                  ADD   R2, R2, R1
                  ADD   R2, R2, #-1     ; point to one's digit

                  LDR   R4, R2, #0      ; load ones digit
                  AND   R4, R4, x0F     ; lowest four bits = value
                  ADD   R0, R0, R4      ; add to result
                  ADD   R1, R1, #-1     ; decrement counter
                  BRz   AtoB_DONE
```

ASCII to Binary Subroutine: Part 2

```
                LDR   R4, R2, #-1     ; load tens digit
                AND   R4, R4, x0F     ; convert to binary
                LEA   R3, Lookup10    ; index into lookup table
                ADD   R3, R3, R4
                LDR   R4, R3, #0
                ADD   R0, R0, R4      ; add to result
                ADD   R1, R1, #-1     ; decrement counter
                BRz   AtoB_DONE

                LDR   R4, R2, #-2     ; load hundreds digit
                AND   R4, R4, x0F
                LEA   R3, Lookup100   ; multiply by 100
                ADD   R3, R3, R4
                LDR   R4, R3, #0
                ADD   R0, R0, R4      ; add to result

AtoB_DONE       LD    R1, AtoB_SR1    ; restore regs and return
                LD    R2, AtoB_SR2
                LD    R3, AtoB_SR3
                LD    R4, AtoB_SR4
                RET
```

ASCII to Binary Subroutine: Part 3

```
AtoB_ASCIIBUFF     .FILL  ASCIIBUFF    ; pointer to string buffer
AtoB_SR1           .BLKW  1
AtoB_SR2           .BLKW  1
AtoB_SR3           .BLKW  1
AtoB_SR4           .BLKW  1
Lookup10           .FILL  #0     ; lookup table for x10
                   .FILL  #10
                   .FILL  #20
                   .FILL  #30
                   .FILL  #40
                   .FILL  #50
                   ; 60-90 go here: removing values to fit on slide...
Lookup100          .FILL  #0
                   .FILL  #100
                   .FILL  #200
                   .FILL  #300
                   .FILL  #400
                   .FILL  #500
                   ; 600-900 go here...
```

Exercise: Make this code safer by checking if entered string is a 3-digit decimal number.

Binary to ASCII

Data Storage: Same buffer as before. Resulting buffer will always have four characters -- a sign (+ or −) and three digits, including leading zeroes.

Computation: Instead of multiplying by 10 or 100, we need to divide.

The lookup table method is not so helpful this time. Our input value is in the range 0 to 999, so we would need 1000 entries in the lookup table.

Instead, we will repeatedly subtract 10 (or 100) until the result goes negative. If we count how many times we subtracted, that is the quotient.

Example: Divide 392 by 100.

Quotient = 0

392 − 100 = 292: ≥ 0, quotient = 1

292 − 100 = 192: ≥ 0, quotient = 2

192 − 100 = 92: ≥ 0, quotient = 3

92 − 100 = −8: < 0, STOP! quotient = 3

Binary to ASCII: Algorithm

R0 is number to be converted.
Initialize pointer (R1) to ASCIIBUFF.

If R0 < 0:

Store minus sign (x2D) to M[R1].

Negate R0 (to make it positive).

Else:

Store plus sign (x2B) to M[R1].

R2 = '0'

Each time we subtract 100 and get a positive value, add +1 to R2.
Store R2 (hundreds digit) to M[R1+1].

R2 = '0'

Each time we subtract 10 and get a positive value, add +1 to R2.
Store R2 (tens digit) to M[R1+2].

R2 = '0' + remainder (ones digit), store to M[R1+3].

Binary to ASCII Subroutine: Part 1

```
; Convert value in R0 to 4-character ASCII buffer: +xxx or -xxx.
; ASCIIBUFF is address of string buffer.

BinaryToASCII     ST    R0, BtoA_SR0     ; save registers
                  ST    R1, BtoA_SR1
                  ST    R2, BtoA_SR2
                  ST    R3, BtoA_SR3
                  LD    R1, BtoA_ASCIIBUFF  ; pointer to buffer

                  ADD   R0, R0, #0       ; check for +/-
                  BRn   NegSign
                  LD    R2, ASCIIPlus    ; store '+'
                  BRnzp  Begin100
NegSign           LD    R2, ASCIINeg     ; store '-'
                  NOT   R0, R0
                  ADD   R0, R0, #1       ; absolute value

Begin100          STR   R2, R1, #0
```

Binary to ASCII Subroutine: Part 2

```
                LD    R2, ASCIIoffset   ; R2 = '0' (hundreds digit)
                LD    R3, Neg100
Loop100         ADD   R0, R0, R3
                BRn   End100
                ADD   R2, R2, #1        ; next digit
                BRnzp  Loop100
End100          STR   R2, R1, #1        ; store hundreds digit char
                LD    R3, Pos100
                ADD   R0, R0, R3        ; restore R0 to positive

                LD    R2, ASCIIoffset   ; R2 = '0' (tens digit)
Loop10          ADD   R0, R0, #-10
                BRn   End10
                ADD   R2, R2, #1        ; next digit
                BRnzp  Loop10
End10           STR   R2, R1, #2        ; store tens digit char
                ADD   R0, R0, #10       ; restore R0 to positive
```

Binary to ASCII Subroutine: Part 3

```
                LD    R2, ASCIIoffset    ; R2 = '0' (ones digit)
                ADD   R2, R2, R0         ; convert to char
                STR   R2, R1, #3         ; store ones digit char

                LD    R0, BtoA_SR0       ; restore regs and return
                LD    R1, BtoA_SR1
                LD    R2, BtoA_SR2
                LD    R3, BtoA_SR3
                RET

ASCIIPlus       .FILL '+'
ASCIINeg        .FILL '-'
ASCIIoffset     .FILL '0'
Neg100          .FILL #-100
Pos100          .FILL #100
BtoA_SR0        .BLKW 1
BtoA_SR1        .BLKW 1
BtoA_SR2        .BLKW 1
BtoA_SR3        .BLKW 1
BtoA_ASCIIBUFF  .FILL ASCIIBUFF
```

Data Conversion Challenges

Challenge #1: ASCII to Binary (Exercise 10.4)

Devise an algorithm that can convert a input strings of arbitrary size.

We can't create an indefinite number of lookup tables, because we don't know the maximum power of 10 needed. Also assume that we don't know the maximum number of bits in a binary integers.

Challenge #2: Binary to ASCII (Exercise 10.6)

Devise an algorithm that does not create unneeded characters. In other words, do not create leading zeroes or a leading '+'. You can still assume that the range of integer values is −999 to +999.

Using a Stack for Arithmetic

While LC-3 and modern ISAs use general-purpose registers for temporary storage, some ISAs use no registers at all. So-called **stack machines** use a memory stack for all source and destination operands.

Example:

- ADD specifies no source or destination operands.
  The instruction always pops two values from the stack, adds them together, and then pushes the result to the stack.

In this architecture, the number of temporary values is limited only by the size of the memory stack, and the programmer never needs to keep track of which register holds which value.

Arithmetic Expressions: Postfix Notation

We are used to writing arithmetic expressions using infix notation, with the operator in between the operands: 100 + 47

For a stack machine, it's more convenient to write the expression using postfix notation, where the operator comes after the relevant operands, like this: 100 47 +

This can be interpreted as: push 100, push 47, ADD
The result (147) is pushed onto the stack when the ADD is complete.

More complex expressions can be handled with more temporary values on the stack.

Infix	*Postfix*
(25 + 17) × (3 + 2)	25 17 + 3 2 + x

There's a reasonably simple algorithm to convert an expression from infix to postfix notation, but it's beyond the scope of this discussion. (But it uses a stack!)

Arithmetic Subroutines

We will need three subroutines for the three operations performed by our calculator.

OpAdd Pop two integers from the stack, add, push the result.

OpMult Pop two integers from the stack, multiply, push the result.

OpNeg Pop one integer from the stack, negate, push the result.

These routines all use the User Stack and the Push/Pop subroutines defined in Chapter 8.

We also create a **RangeCheck** subroutine, to make sure that an integer value (in R0) is within the specified range of −999 to +999. If out of range, an error message is printed and a value of 1 is returned in R5. Otherwise, 0 is returned in R5.

OpAdd Subroutine

```
OpAdd              ; save regs: R0, R1, R5, R7 (omitted to save space)
                   JSR   POP          ; pop first arg into R0
                   ADD   R5, R5, #0   ; check for success
                   BRp   OpAdd_EXIT

                   ADD   R1, R0, #0   ; move to R1
                   JSR   POP          ; pop second arg into R0
                   ADD   R5, R5, #0   ; check for success
                   BRp   OpAdd_Restore1   ; if fail, put 1st back on stack

                   ADD   R0, R0, R1   ; perform the add
                   JSR   RangeCheck   ; check result
                   ADD   R5, R5, #0
                   BRp   OpAdd_Restore2   ; if fail, put both args back
                   JSR   PUSH         ; push result

OpAdd_Restore2     ADD   R6, R6, #-1
OpAdd_Restore1     ADD   R6, R6, #-1
OpAdd_Exit         ; restore regs (omitted to save space)
                   RET
```

RangeCheck Subroutine

```
; R5 = 0 if -999 <= R0 <= +999; otherwise, R5 = 1
RangeCheck        LD    R5, Neg999
                  ADD   R5, R0, R5
                  BRp   BadRange    ; R0 > 999
                  LD    R5, Pos999
                  ADD   R5, R0, R5
                  BRn   BadRange    ; R0 < -999
                  AND   R5, R5, #0
                  RET
BadRange          ST    R0, RangeCheck_SR0
                  LEA   R0, RangeErrorMsg
                  PUTS
                  AND   R5, R5, #0
                  ADD   R5, R5, #1
                  LD    R0, RangeCheck_SR0
                  RET
Neg999            .FILL #-999
Pos999            .FILL #999
RangeErrorMsg     .FILL x000A  ; linefeed
                  .STRINGZ "Error: Number is out of range."
RangeCheck_SR0    .BLKW 1
```

OpMult and OpNegate

Subroutines are similar to OpAdd.

OpMult -- see Figures 10.11 and 10.12

OpNeg -- see Figure 10.13

Complete Program: User Interface

Program will print a prompt for the user to enter a command.

If the command is X (exit), the machine halts.
Otherwise, the command is performed and the program prompts for another command. Each command will be echoed as the user types.

Cmd Character	Description
X	Exit the program. (Halts the machine.)
C	Clear the stack. All temporary values are removed.
+	ADD -- pop two values, add, push result.
*	MULTIPLY -- pop two values, multiply, push result.
–	NEGATE -- pop one value, negate, push result.
LF/CR	User types a positive decimal integer, up to 3 digits, followed by linefeed (or Enter). Value is pushed to the stack.

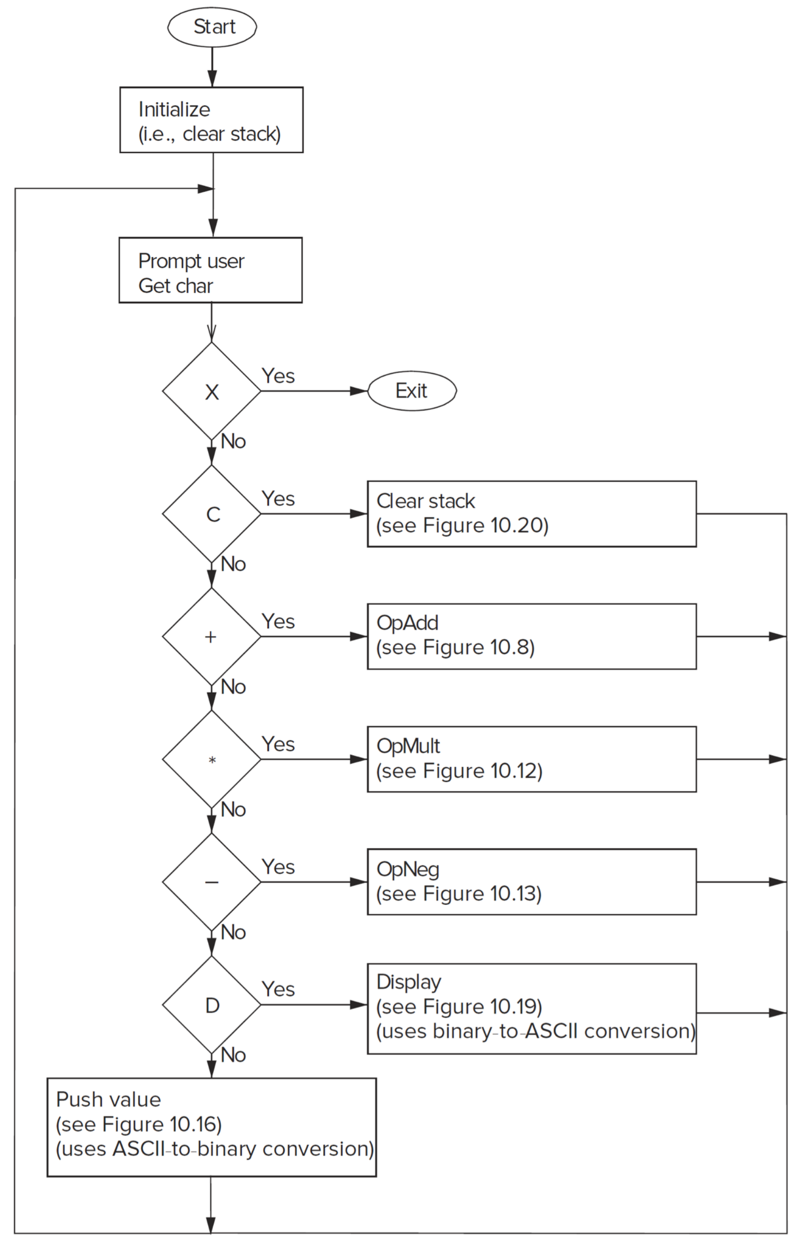
Program Flowchart

The flow of the main program is shown to the right. Many subroutines have been defined, but we have three more:

OpClear Clear the stack.

OpDisplay Convert top of stack to ASCII and print.

PushValue Convert user input to binary and push.



Access the text alternative for slide images.

OpCear Subroutine

```
; Clear the stack -- set R6 (stack pointer) to its max value
; Main program defines StackBase as max. address allocated to stack

OpClear            LD   R6, OpClear_StackBase
                   ADD  R6, R6, #1   ; empty = one beyond highest addr
                   RET
OpClear_StackBase .FILL StackBase
```

OpDisplay Subroutine

```
; Convert top of stack to ASCII -- store chars in ASCIIBUFF
; (Do not pop the stack.)  Print linefeed after string.

OpDisplay          ST   R0, OpDisplay_SR0   ; save regs
                   ST   R5, OpDisplay_SR5
                   ST   R7, OpDisplay_SR7
                   JSR  POP                ; pop stack into R0
                   ADD  R5, R5, #0         ; check for error
                   BRp  OpDisplay_DONE
                   JSR  BinaryToASCII      ; convert R0 to ASCII
                   LD   R0, NewlineChar
                   OUT
                   LD   R0, OpDisplay_ASCIIBUFF
                   PUTS
                   ADD R6, R6, #-1 ; push displayed number back on stack
OpDisplay_DONE     ; restore regs (omitted for space)
                   RET
NewlineChar        .FILL x000A
OpDisplay_ASCIIBUFF  .FILL ASCIIBUFF
OpDisplay_SR0      .BLKW 1
OpDisplay_SR5      .BLKW 1
OpDisplay_SR7      .BLKW 1
```

PushValue Subroutine: Part 1

```
; Called when command is not X, C, +, *, or -.
; Expect user input to be 0-3 digits followed by linefeed.
; Convert to binary.
PushValue         ; save R0, R1, R2, R7 (omitted for space)
                  LD    R1, PV_ASCIIBUFF   ; ptr to string buffer
                  LD    R2, MaxDigits      ; will count down to check # digits

ValueLoop         ADD   R3, R0, #-10       ; if linefeed
                  BRz   GoodValue
                  ADD   R2, R2, #0         ; if 4th digit, bad value
                  BRz   TooLargeInput
                  LD    R3, NegASCII0      ; check for digit
                  ADD   R3, R0, R3
                  BRn   NotInteger
                  LD    R3, NegASCII9
                  ADD   R3, R0, R3
                  BRp   NotInteger
                  ADD   R2, R2, #-1        ; count digit and store
                  STR   R0, R1, #0
                  ADD   R1, R1, #1
                  GETC                     ; read and echo next char
                  OUT
                  BRnzp ValueLoop
```

PushValue Subroutine: Part 2

```
GoodInput          LD    R2, PV_ASCIIBUFF
                   NOT   R2, R2
                   ADD   R2, R2, #1
                   ADD   R1, R1, R2            ; calculate # digits
                   BRz   NoDigit
                   JSR   ASCIItoBinary         ; push binary value to stack
                   JSR   PUSH
                   BRnzp PushValue_DONE

NoDigit            LEA   R0, NoDigitMsg
                   PUTS
                   BRnzp PushValue_DONE

NotInteger         GETC                       ; read chars until linefeed
                   OUT
                   ADD   R0, R0, #-10
                   BRnp  NotInteger
                   LEA   R0, NotIntegerMsg
                   PUTS
                   BRnzp PushValue_DONE
```

PushValue Subroutine: Part 3

```
TooLargeInput     GETC                          ; get chars until linefeed
                  OUT
                  ADD  R0, R0, #-10
                  BRnp TooLargeInput
                  LEA  R0, TooManyDigits
                  PUTS

PushValue_DONE    ; restore regs (omitted for space)
                  RET

TooManyDigits     .FILL x000A     ; linefeed before message
                  .STRINGZ "Too many digits"
NotIntegerMsg     .FILL x000A
                  .STRINGZ "Not an integer"
NoDigitMsg        .FILL x000A
                  .STRINGZ "No digits entered"
MaxDigits         .FILL #3
NegASCII0         .FILL x-30
NegASCII9         .FILL x-39
PV_ASCIIBUFF      .FILL ASCIIBUFF
```

Program Main Routine: Part 1

```
; stack of 10 values is allocated at end of main routine
                    LEA   R6, StackBase
                    ADD   R6, R6, #1        ; empty stack

NewCommand          LEA   R0, PromptMsg     ; print prompt
                    PUTS
                    GETC                    ; get command character and echo
                    OUT

TestX               LD    R1, NegX    ; check for X
                    ADD   R1, R1, R0
                    BRnp  TestC       ; if no match, go to next command
                    HALT

TestC               LD    R1, NegC    ; check for C
                    ADD   R1, R1, R0
                    BRnp  TestAdd
                    JSR   OpClear
                    BRnzp NewCommand

                    ; similar code (omitted) for +, *, -, D
```

Program Main Routine: Part 2

```
; if reaches this point, none of the other commands have matched
; should be a number

EnterNumber        JSR   PushValue
                   BRnzp NewCommand

PromptMsg          .FILL x000A
                   .STRINGZ "Enter a command: "
NegX               .FILL xFFA8    ; negative of 'X'
NegC               .FILL xFFBD    ; negative of 'C'
                   ; etc.

StackMax           .BLKW 9     ; space for stack
StackBase          .BLKW 1
ASCIIBUFF          .BLKW 4
                   .FILL x0000 ; null terminator for display string
```

Because learning changes everything.®

www.mheducation.com