Program Flow Charting

How to tackle the beginning stage of a program design
A Program

Set of instructions written in a programming language that tells the computer what to do
Programmers

• Prepare instructions that make up the program
• Run the instructions to see if they produce the correct results
• Make corrections
• **Document the program**
• Interact with
  – Users
  – Managers
  – Systems analysts
• Coordinate with other programmers to build a complete system
The Programming Process

- Defining the problem
- Planning the solution
- Coding the program
- Testing the program
- Documenting the program
The Programming Process: 

*Defining the Problem*

- What is the input
- What output do you expect
- How do you get from the input to the output
The Programming Process:

*Planning the Solution*

- **Algorithms**
  - Detailed solutions to a given problem
    - Sorting records, adding sums of numbers, etc.

- **Design tools**
  - Flowchart
  - Pseudocode
    - Has logic structure, but no command syntax
The Programming Process: Planning the Solution

- Desk-checking
  - Personal code design walk through
- Peer Reviews
  - “Code walk through”/structured walk through
Flow Control Elements
Accept series of numbers and display the average
The Programming Process: Coding the Program

- Translate algorithm into a formal programming language
- Within syntax of the language
- How to key in the statements?
  - Text editor
  - Programming environment
    - Interactive Development Environment (IDE)
    - MPL06X
The Programming Process:

**Testing the Program**

- Translation – compiler
  - Translates from source module into object module
  - Detects syntax errors

- Link – linkage editor (linker)
  - Combines object module with libraries to create load module
  - Finds undefined external references

- Debugging
  - Run using data that tests all statements
  - Logic errors
The Programming Process: Documenting the Program

- Performed throughout the development
- Material generated during each step
  - Problem definitions
  - Program plan
  - Comments within source code
  - Testing procedures
  - Narrative
  - Layouts of input and output
  - Program listing
Procedural Level Languages

- 1\textsuperscript{st} Generation: Machine Level
- 2\textsuperscript{nd} Generation: Assembly Level
- 3\textsuperscript{rd} Generation: High Level
FORTRAN

```fortran
C FORTRAN PROGRAM
C AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD
WRITE (6,10)
SUM = 0
COUNTER = 0
WRITE (6,60)
READ (5,40) NUMBER
1 IF (NUMBER .EQ. 999) GOTO 2
SUM = SUM + NUMBER
COUNTER = COUNTER + 1
WRITE (6,70)
READ (5,40) NUMBER
GO TO 1
2 AVERAGE = SUM / COUNTER
WRITE (6,80) AVERAGE
10 FORMAT (1X, 'THIS PROGRAM WILL FIND THE AVERAGE OF',
     & 'INTEGERS YOU ENTER ',/1X, 'THROUGH THE',
     & 'KEYBOARD. TYPE 999 TO INDICATE END OF DATA.',/)
40 FORMAT (1X)
60 FORMAT (1X, 'PLEASE ENTER A NUMBER ')
70 FORMAT (1X, 'PLEASE ENTER THE NEXT NUMBER ')
80 FORMAT (1X, 'THE AVERAGE OF THE NUMBERS IS ',F6.2)
STOP
END
```

This program will find the average of integers you enter through the keyboard. Type 999 to indicate end of data.

Please enter a number 6
Please enter the next number 4
Please enter the next number 11
Please enter the next number 999

The average of the numbers is 7.00
The program in COBOL below is designed to find the average of integers entered through the keyboard. It uses the following sections:

**Identification Division**
- Program-name: AVERAGE
- Environment Division

**Configuration Section**
- Source-computer: H EP 9000
- Object-computer: H EP 9000

**Working-STorage Section**
- O1 AVERAGE PICTURE ---9.99
- O1 COUNTER PICTURE 99999
- O1 NUMBER-ITDH PICTURE 99999
- O1 SUM-ITDH PICTURE 99999
- O1 BLANK-LINE VALUE SPACE

**Procedure Division**

100-Control-Routine
- Perform 200-Display-Instructions
- Perform 300-Initialization-Routine
- Perform 400-Enter-And-Add
- Until (number-itdh = 999)
- Perform 500-Calculate-Average
- Perform 600-Display-Results

Stop

200-Display-Instructions
- Display "This program will find the average of integers you enter"
- Display "through the keyboard. Type 999 to indicate end of data.
- Display Blank-Line

300-Initialization-Routine
- Display "Please enter a number"
- Accept number-itdh
- 400-Enter-And-Add
- Add number-itdh to sum-itdh
- Add 1 to counter
- Display "Please enter the next number"
- Accept number-itdh
- 500-Calculate-Average
- Divide sum-itdh by counter giving average
- 600-Display-Results
- Display "The average of the numbers is ".average

(a) This program will find the average of integers you enter through the keyboard. Type 999 to indicate end of data.

(b) Please enter a number
   6
   Please enter the next number
   4
   Please enter the next number
   11
   Please enter the next number
   999
   The average of the numbers is 7.00
Third Generation Languages

```basic
' BASIC PROGRAM
' AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD
CLS
PRINT "THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER"
PRINT "THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA."
PRINT
SUM=0
COUNTER=0
PRINT "PLEASE ENTER A NUMBER"
INPUT NUMBER
DO WHILE NUMBER <> 999
    SUM=SUM+NUMBER
    COUNTER=COUNTER+1
    PRINT "PLEASE ENTER THE NEXT NUMBER"
    INPUT NUMBER
LOOP
AVERAGE=SUM/COUNTER
PRINT "THE AVERAGE OF THE NUMBERS IS"; AVERAGE
END
```

(a)

```
THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA.

PLEASE ENTER A NUMBER
?6
PLEASE ENTER THE NEXT NUMBER
?4
PLEASE ENTER THE NEXT NUMBER
?11
PLEASE ENTER THE NEXT NUMBER
?999
THE AVERAGE OF THE NUMBERS IS 7
```

(b)
// C++ PROGRAM
// AVERAGING INTEGERS ENTERED THROUGH THE KEYBOARD

#include <iostream.h>
main ()
{
    float average;
    int number, counter = 0; int sum = 0;
    cout << "THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER\n";
    cout << "THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA. \n";
    cout << "PLEASE ENTER A NUMBER: \n";
    cin >> number;
    while (number != 999)
    {
        sum += sum + number;
        counter ++;
        cout << "\nPLEASE ENTER THE NEXT NUMBER":
        cin >> number;
    }
    average = sum / counter;
    cout << \nTHE AVERAGE OF THE NUMBERS IS " << average

C++

THIS PROGRAM WILL FIND THE AVERAGE OF INTEGERS YOU ENTER THROUGH THE KEYBOARD. TYPE 999 TO INDICATE END OF DATA.
PLEASE ENTER A NUMBER 6
PLEASE ENTER THE NEXT NUMBER 4
PLEASE ENTER THE NEXT NUMBER 11
PLEASE ENTER THE NEXT NUMBER 999
THE AVERAGE OF THE NUMBERS IS 7.00
LC-3
Assembly Language
(Ch7)
LC-3 is a load/store RISC architecture

- Has 8 general registers
- Has a flat 16-bit addressing range
- Has a 16-bit word size
- Load variables from memory to register
Syntax of LC-3

- One instruction, declaration per line
- Comments are anything on a line following ";"
- Comments may not span lines

LC-3 has 2 basic data types
- Integer
- Character

Both take 16-bits of space (a word) though a character is only 8-bits in size.
Labels

- Symbolic names that are used to identify memory locations
- Location for target of a branch or jump
- Location for a variable for loading and storing
- Can be 1-20 characters in size

PC Offset + 9
Br Done
**Directives** give information to the assembler. All directives start with ‘.’ (period)

\( \text{\texttt{0x3000}} \)

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>Where to start in placing things in memory</td>
</tr>
<tr>
<td>.FILL</td>
<td>Declare a memory location</td>
</tr>
<tr>
<td>.BLKW</td>
<td>Reserve a group of memory locations</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>Declare a group of characters in memory</td>
</tr>
<tr>
<td>.END</td>
<td>Tells assembly where your program source ends</td>
</tr>
</tbody>
</table>
.ORIG

• Tells simulator where to put your code in memory
• Only one allowed per program
• PC gets set to this address at start up
• Similar to the “main” in “C”
"C"

```c
    type  varname;
```

type is

- `int` (integer)
- `char` (character)
- `float` (floating point)

"LC-3"

```c
    varname  .FILL  value
```

value is required -- the initial value
LC-3 Syntax

.FILL

flag .FILL x0001
counter .FILL x0002
letter .FILL x0041 ; A
letters .FILL x4241 ; BA

• One declaration per line
• Always declaring 16-bits, the word size of LC-3
• Don’t mix in with your code, will be treated like an instruction
.org

Code

Halt

Fill

.end

Trap x25
.BLKW

- Tells assembler to set aside some number of sequential memory locations
- Useful for arrays
- Can be initialized
Examples of .BLKW:

; set aside 3 locations
.BLKW 3

; set aside 1 location and label it.
Bob .BLKW 1

; set aside 1 location, label and initialize to 4.
Num .BLKW 1 #4

Bob +2

bob
LC-3 Syntax

.STRNGZ

- Used to declare a string of characters
- Is terminated by x0000
- One character per memory location

Example:

```
hello .STRNGZ "Hello World!"
```
LC-3 Syntax

.END

- Tells the assembler where your program ends
- Only one allowed in your program
### LC-3 Syntax

<table>
<thead>
<tr>
<th>“LC-3”</th>
<th>“C”</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1, X</td>
<td></td>
</tr>
<tr>
<td>LD R2, Y</td>
<td></td>
</tr>
<tr>
<td>ADD R3, R2, #0</td>
<td>Z = Y</td>
</tr>
<tr>
<td>ADD R3, R1, R2</td>
<td>Z = X + Y</td>
</tr>
<tr>
<td>???</td>
<td>Z = X - Y</td>
</tr>
<tr>
<td>???</td>
<td>Z = X * Y</td>
</tr>
<tr>
<td>???</td>
<td>Z = X / Y</td>
</tr>
<tr>
<td>ST R3, Z</td>
<td></td>
</tr>
</tbody>
</table>

An immediate is a value specified in an instruction, not by a .FILL declaration
Simple LC-3 program

```assembly
.ORIG x3000
LD R2, Zero
LD R0, M0
LD R1, M1
BRz
ADD R2, R2, R0
ADD R1, R1, -1
BR
ST
HALT
.FILL x0000
.FILL x0000
.FILL x0004
.FILL x0002
.END
```

- What does this program do?
- What is in “Result” at the end?
\[ x \cdot y = \sum_{n=1}^{\infty} x \]

<table>
<thead>
<tr>
<th></th>
<th>R2</th>
<th>R1</th>
<th>RO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
Program Execution

• Assembler translates to executable – machine language
• Linker combines multiple LC-3 files – if any
• Loader puts executable into memory and makes the CPU jump to first instruction, .ORIG.
• Executes
• When executing is done returns control to OS
  • Or simulator or monitor
• Load again to run again with different data
  • In this case, assemble again, too, since data is in program.
HLL – if/else statements...

if (condition)
    statement;
else
    statement;
“C”  
if (count < 0) 
    count = count + 1;

“LC-3”  
LD      R0, count
BRpz   greatzero
ADD     R0, R0, #1
greatzero  ; next instruction goes here
Loops can be built out of IF’s – WHILE:

```
“C”

while (count > 0)
{
    a = a + count;
    count--;
}
```
"LC-3"

LD R1, a
LD R0, count
while BRnz
ADD R1, R1, R0
ADD R0, R0, #-1
BR
while
ST
ST
 endwhile
 endwhile
R0, count
Procedure Calls

Simple procedure calls require 2 instructions:

“JSR” or “JSRR” Jump Service Routine
  • Saves the return address into R7

“RET” Jump Return
  • Be careful with registers!!
  • Cannot nest unless R7 is saved elsewhere
  • Cannot be recursive without a stack
Example

JSR Sub ; calls procedure
...

; calculate R2 = R0 - R1
Sub
  NOT
  ADD
  ADD
  RET
  R2, R1
  R2, R2, #1
  R2, R2, R0
; returns to line after
; JSR Sub
Repeat loops

"C"

/* do statement while expression is TRUE */
/* when expression is FALSE, exit loop */
do {
    if (a < b)
        a++;  
    if (a > b)
        a--;  
} while (a != b)
“LC-3”

repeat

secondif

until

LD R0, a
LD R1, b
JSR Sub ; R2 = R0-R1
BRpz secondif
ADD R0, R0, #1
JSR Sub
BRnz until
ADD R0, R0, #-1
JSR Sub
BRnp repeat
For loops

```
for ( l = 3; l <= 8; l++)
{ a = a+l; }
```
```
"LC-3"

; R0=a, R1=l, R2=temp

LD   R0, a
AND  R1, R1, #0    ; init l to zero
ADD  R1, R1, #3    ; now make 3

for

ADD  R2, R1, #-8

BRp  endfor

ADD  R0, R0, R1    ; a=a+l
ADD  R1, R1, #1    ; l++

BR   for

endfor
```
TRAP
(System Calls)

• Very tedious and dangerous for a programmer to deal with IO at the OS level.
• Need an instruction though to get the attention of the OS.

Use the “TRAP” instruction and a “trap vector”.

<table>
<thead>
<tr>
<th>Trap Vector</th>
<th>Assembler Name</th>
<th>Usage &amp; Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>GETC</td>
<td>Read a character from console into R0, not echoed.</td>
</tr>
<tr>
<td>0x21</td>
<td>OUT</td>
<td>Write character in R0 to console.</td>
</tr>
<tr>
<td>0x22</td>
<td>PUTS</td>
<td>Write string of characters to console. Start with character at address contained in R0. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x23</td>
<td>IN</td>
<td>Print a prompt to console and read in a single character into R0. Character is echoed.</td>
</tr>
<tr>
<td>0x24</td>
<td>PUTSP</td>
<td>Write a string of characters to console, 2 characters per address location. Start with characters at address in R0. First [7:0] and then [15:0]. Stops when 0x0000 is encountered.</td>
</tr>
<tr>
<td>0x25</td>
<td>HALT</td>
<td>Halt execution and print message to console.</td>
</tr>
</tbody>
</table>

*a b c

A Trap x25
To print a character
  ; the char must be in R0.
TRAP     x21
  
or
OUT

To read in a character
  ; will go into R0, no echo.
TRAP     x20
  
or
GETC
To end your program:

TRAP

or

HALT

x25