11 ARM Programming 2
CSC 230

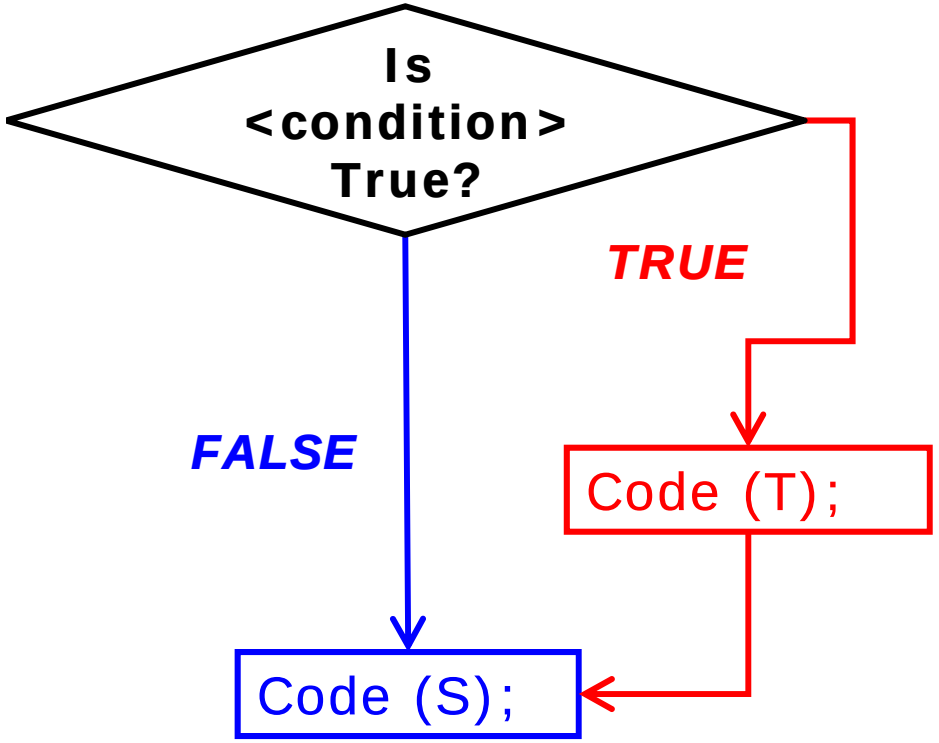
Department of Computer Science
University of Victoria

Stallings chapters 12,13 (skip Intel portions)
M&H: chapter 4 translated from ARC
ARM Manual

Control Structures

- IF ... THEN
- IF ... THEN ... ELSE
- WHILE ... DO
- FOR ...
- DO ... WHILE (a.k.a. REPEAT)

Control Structures: IF ... THEN



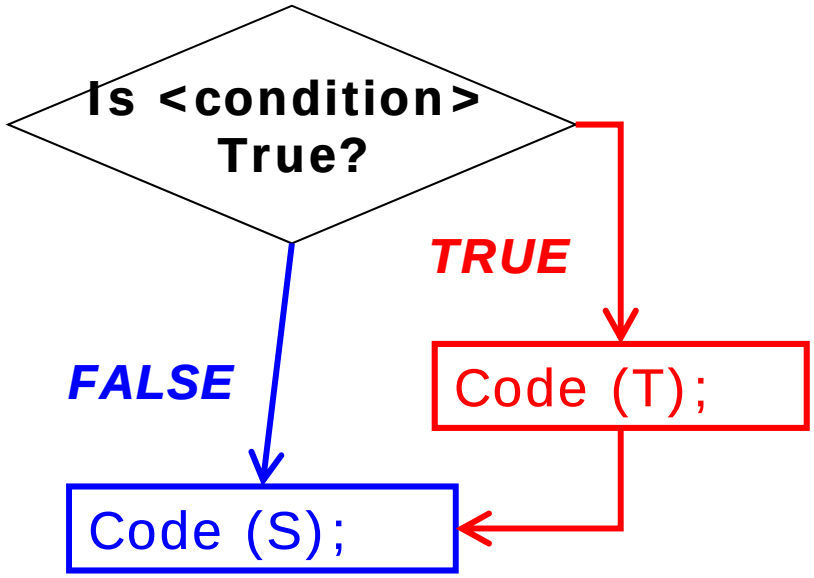
if TRUE then execute:

Code (T);

Code (S);

if FALSE then execute:

Code (S) only;

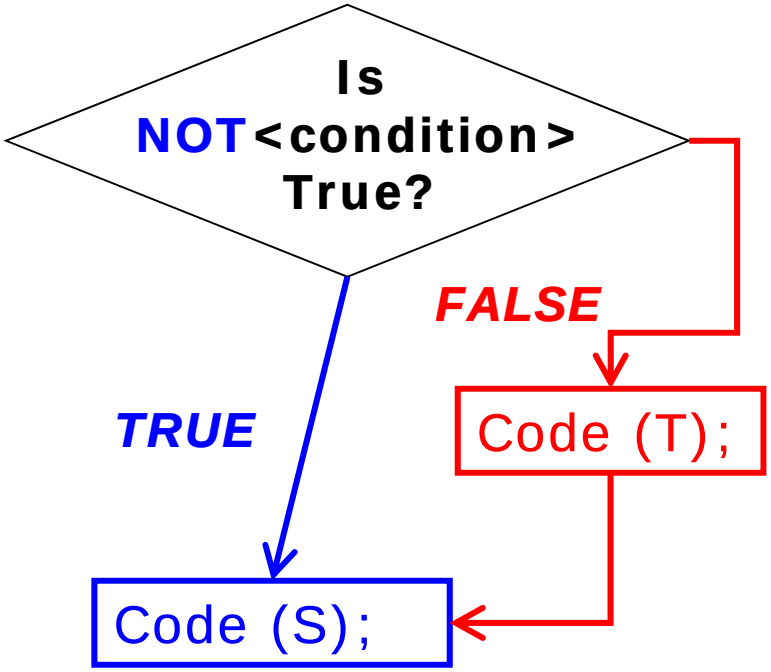



```
IF (X = = Y) {
    Z = X + Z;} /*CodeT*/
K = Z – T;      /*CodeS*/
```

```
          CMP  X,Y
          BEQ  CodeT
          BNE  CodeS
CodeT:    ADD  R3,R1,R3
CodeS:    SUB  R4,R3,R5
```

if X = = Y

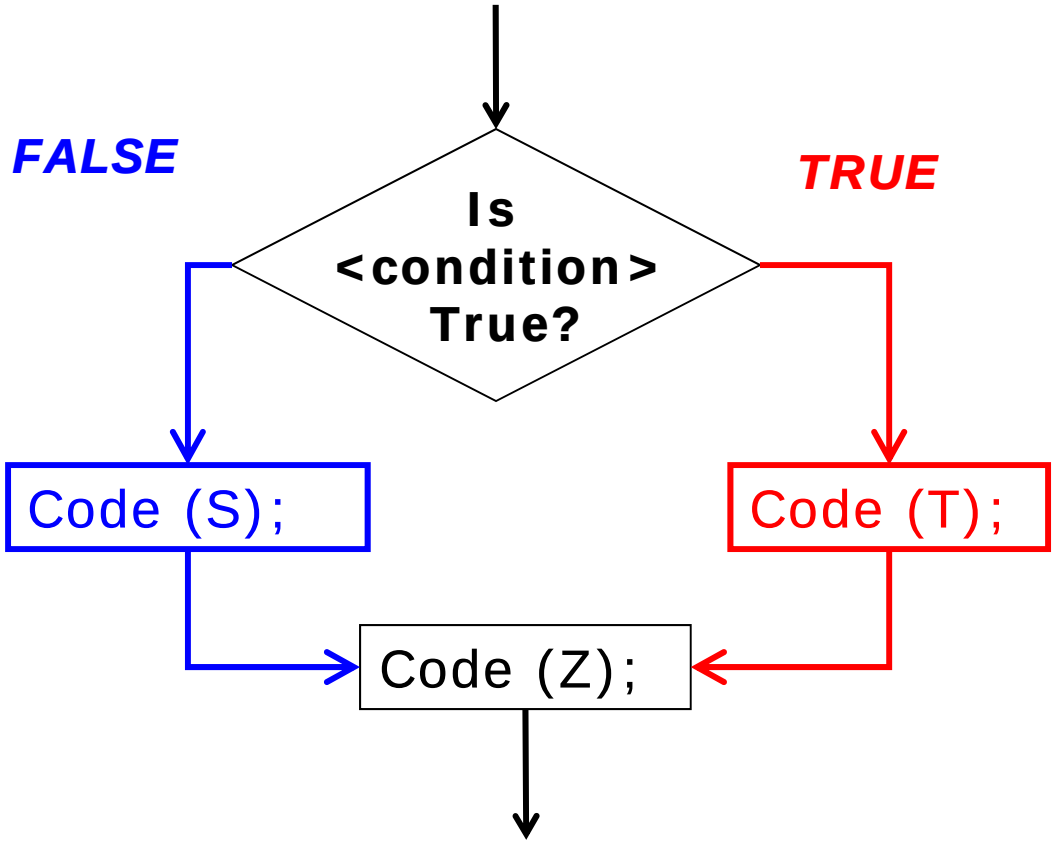
if X != Y




```
IF (X != Y)
    { GO TO CodeS}   break
ELSE Z = X + Z;    /*CodeT*/
CodeS: K = Z – T;      /*CodeS*/
```

```
          CMP  X,Y
          BNE  CodeS
CodeT:    ADD  R3,R1,R3
CodeS:    SUB  R4,R3,R5
```

Control Structures: IF ... THEN ... ELSE



if TRUE then execute:

Code (T);

Code (Z);

ELSE then execute:

Code (S);

Code (Z);

```
if (x > 0) {
    T1 = x – 2;
    }
else {
    S2 = x + 3;
    }
sum=Y+Z;
```

```
    LDR   r1,=x     @r1 = address of x
    LDR   r2,[r1]   @ r2 = x

    CMP   r2,#0     @ compare x to 0
    BGT   T1updt    @ if x>0, goto T1updt
S2updt:
    LDR   r4,=S2    @ r4 = address of S2
    ADD   r5,r2,#3  @ r5 = S2 = x+3
    STR   r5,[r4]   @ store S2
    BAL   ContSum
T1updt:
    LDR   r3,=T1    @ r3=address of T1
    SUB   r6,r2,#2  @ r6 = T1 = x-2
    STR   r6,[r3]   @ store T1
ContSum:
    LDR   r1,=sum   @ r1 = address of sum
    etc. etc.
```

*When **T1updt** is finished, continue directly with code from **ContSum***

*When **S2updt** is finished, need a BAL (a GOTO) to **ContSum** to skip **T1updt***

Test yourself

Do by yourself

R1 = 0x0000 00FF R2 = 0x00000000

CMP R1,R2 R1 – R2 and update CPSR

Which conditional instruction will cause a branch?

BEQ	Equal (zero)	$Z=1$
BNE	Not equal (zero)	$Z=0$
BMI	Minus (negative)	$N=1$
BPL	Plus (positive or zero)	$N=0$
BVS	Overflow	$V=1$
BHI	Unsigned higher	$\overline{C} \vee Z = 0$
BLS	Unsigned lower or same	$\overline{C} \vee Z = 1$
BGE	Signed greater or equal	$N \oplus V = 0$
BLT	Signed less	$N \oplus V = 1$
BGT	Signed greater	$Z \vee (N \oplus V) = 0$
BLE	Signed less	$Z \vee (N \oplus V) = 1$
BAL	Always	

Test yourself

Do by yourself

R1 = 0x0000 00FF R2 = 0x00000000

CMP R1,R2

Which conditional instruction will cause a branch?

BEQ	Equal (zero)	Z=1	No
BNE	Not equal (zero)	Z=0	Yes
BMI	Minus (negative)	N=1	No
BPL	Plus (positive or zero)	N=0	Yes
BVS	Overflow	V=1	No?
BHI	Unsigned higher	$\overline{C} \vee Z = 0$	Yes
BLS	Unsigned lower or same	$\overline{C} \vee Z = 1$	No
BGE	Signed greater or equal	$N \oplus V = 0$	Yes
BLT	Signed less	$N \oplus V = 1$	No
BGT	Signed greater	$Z \vee (N \oplus V) = 0$	Yes
BLE	Signed less	$Z \vee (N \oplus V) = 1$	No
BAL	Always		Yes

Control Structures: loops, choices, details

- conditional or counted
- test at beginning or at end of loop
- increment at beginning or end of loop
- avoid random placement of loop elements
- be careful with nested loops
- watch for register conflicts

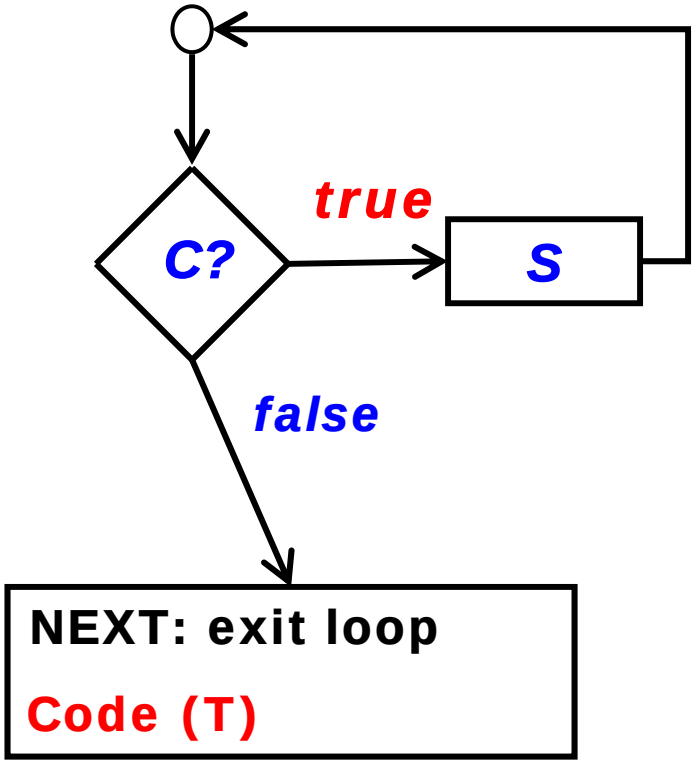
All control and data structures must be implemented by the programmer using low-level primitives (i.e. Branch instructions)

Great care and discipline is required

The main job of a compiler is to translate high-level concepts/control structures to very efficient low-level machine instructions

Control Structures: WHILE ... DO

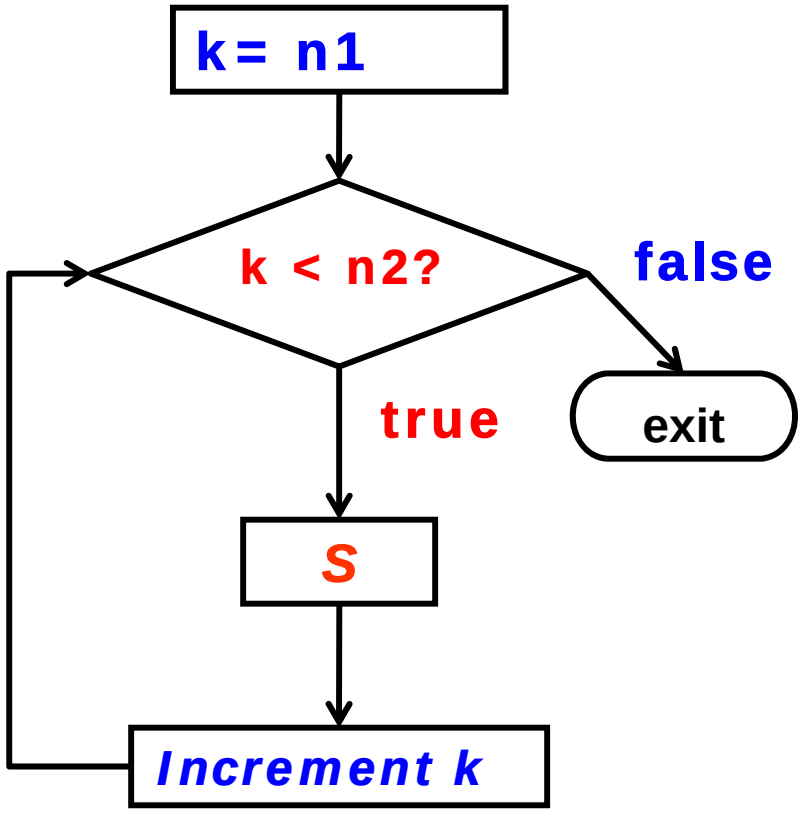
WHILE *C is true* { DO *S* }




```
while (p !=q ) {
        Code (S);
        Modify condition;
   }
Code (T);

Assume R1 =p and R2=q:


W1:   CMP   R1,R2
      BEQ   NEXT
      Code (S);
      (Change R1 or R2)
      BAL   W1
NEXT:Code (T);
```

Control Structures: FOR . . .

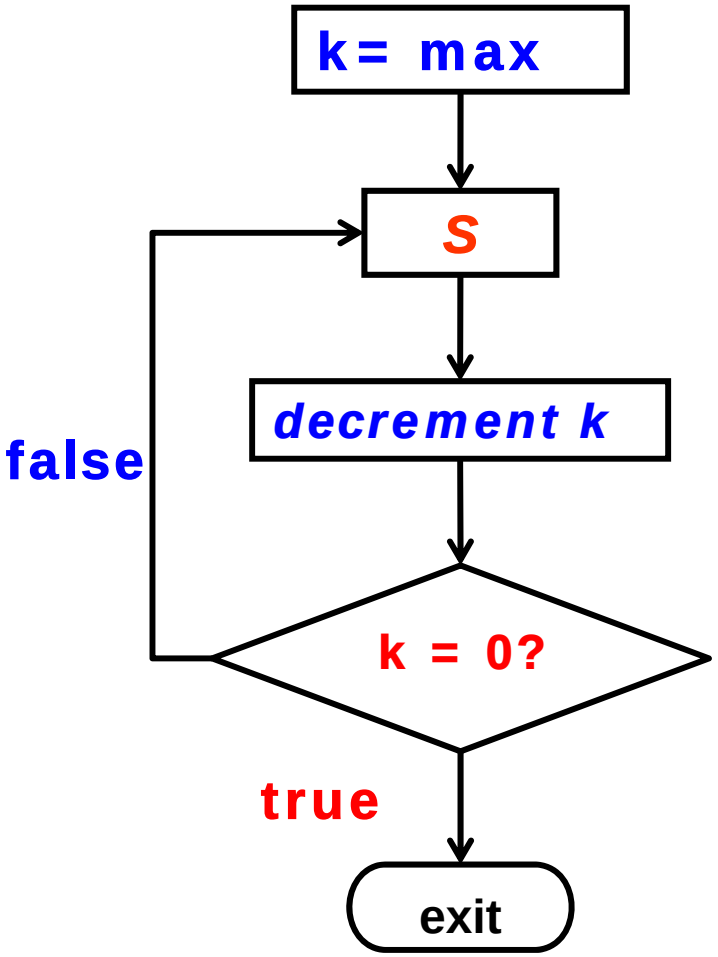
FOR k = n1 to n2-1 DO S




```
for(k=0;k<100;k++){
      Code (S)
    }

@Use register R1 as
@loop counter k

      MOV   R1,#0
F1:   CMP   R1,#100
      BGE   L2
      Code (S)
      ADD   R1,R1,#1
      BAL   F1
L2: loop exit, more code
```

Control Structures: *FOR loop variation: counting down*

FOR k = max downto 0 DO S

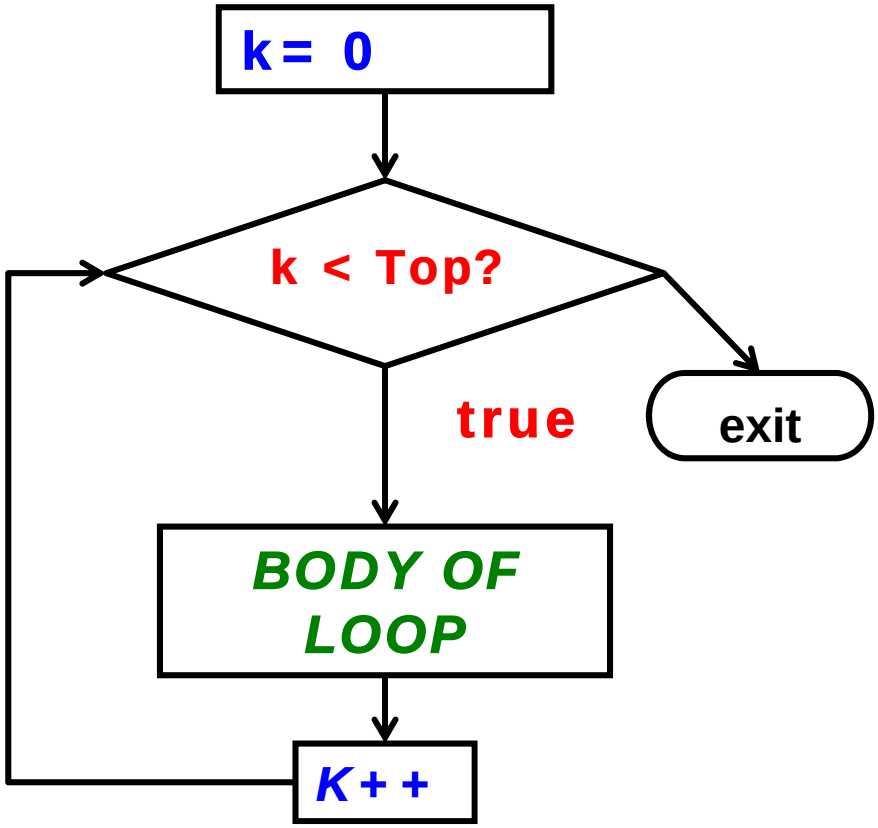



```
for(k=100;k>0;k--){
      Code (S)
    }

@Use register R1 as
@loop counter k

      MOV  R1,#100
F1:   Code (S)
      SUBS R1,R1,#1
      BNE  F1
L2: loop exit, more code
```

A For loop is really a While loop ➔ the While structure is better mapped to assembly language

```
For (k=o; k<Top; k++) {
        BODY OF LOOP
}
```

```
k=o
While (k<Top) {
        BODY OF LOOP
        K++
}
```



Strong Advice

- ✓ ***Avoid distributing the loop control throughout the loop***
  It is best at the beginning or end of the loop

- ✓ ***Avoid unstructured branches from within a loop***
  Only use the equivalent of the break in C/Java;
  i.e. a branch to the instruction immediately following the loop.

- ✓ ***Never branch back in the code from within a loop. The only branches that go backward in the code should be those that return to the top to 'complete' a loop.***
  Only use the equivalent of the continue in C/Java;
  i.e. a branch to the test at the top of the loop.

Counted Loops (FOR): which one?

Counting up from 1 to 100

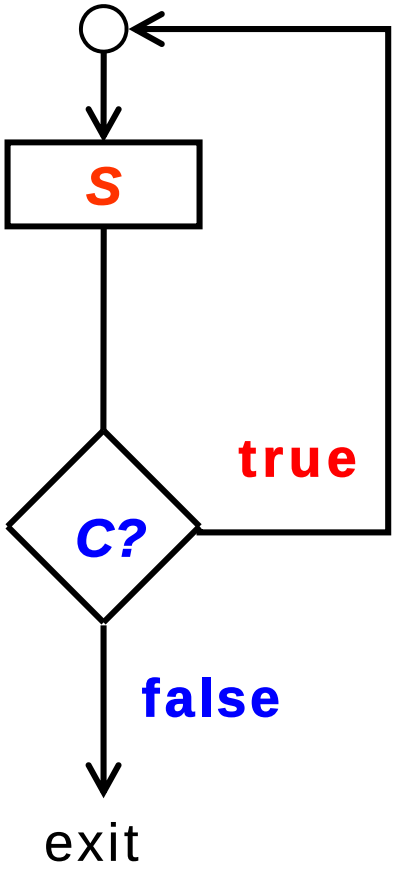
```
      MOV   R1,#1
L1:   :
      :
      ADD   R1,R1,#1
      CMP   R1,#100
      BLT   L1
```

Counting down from 100 to 1

```
      MOV   R1,#100
L1:   :
      :
      SUBS  R1,R1,#1
      BNE   L1
```

Control Structures: DO . . . WHILE (a.k.a. REPEAT)

DO { *S* } WHILE (*C is true)*



➔ **aka REPEAT UNTIL**

A loop that goes through the odd numbers from 1 to 99_{10}

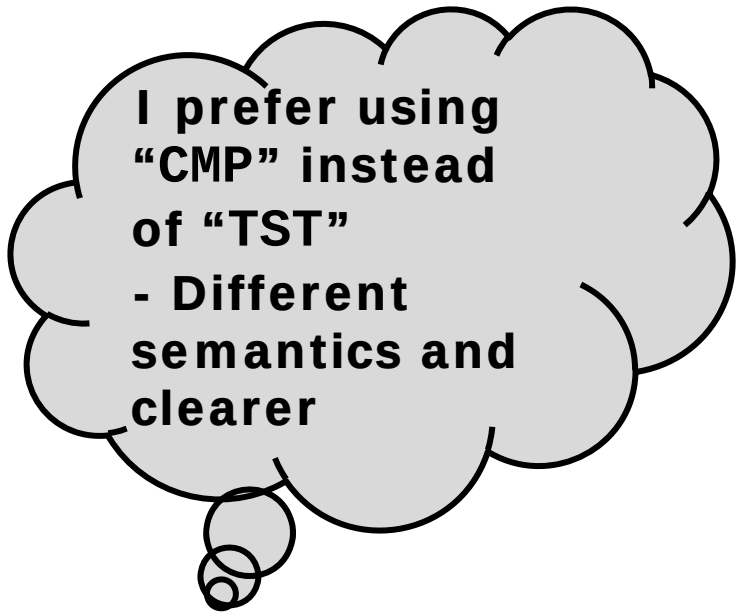
```
      MOV   R1,#1
L1:   :
      :
      ADD   R1,R1,#2
      CMP   R1,#99
      BLE   L1
```

Better?

```
MOV   R1,#98
L1:   :
      :
      SUBS  R1,R1,#2
      BNE   L1
```

Conditional Loops: loop through a string, char by char.

```
        LDR   R2,=STR
L1:     LDRB  R1,[R2]
        TST   R1,#0xFF
        BEQ   L2
        :
        ADD   R2,R2,#1
        BAL   L1
L2:
```



What is the difference with this second version?

```
        LDR   R2,=STR
L1:     LDRB  R1,[R2]
        :
        :
        ADD   R2,R2,#1
        TST   R1,#0xFF
        BNE   L1
```

Example: finding the maximum element in a 1 dimensional array (Quiz 3, 2 years ago)

Write a program to determine the largest integer in an array

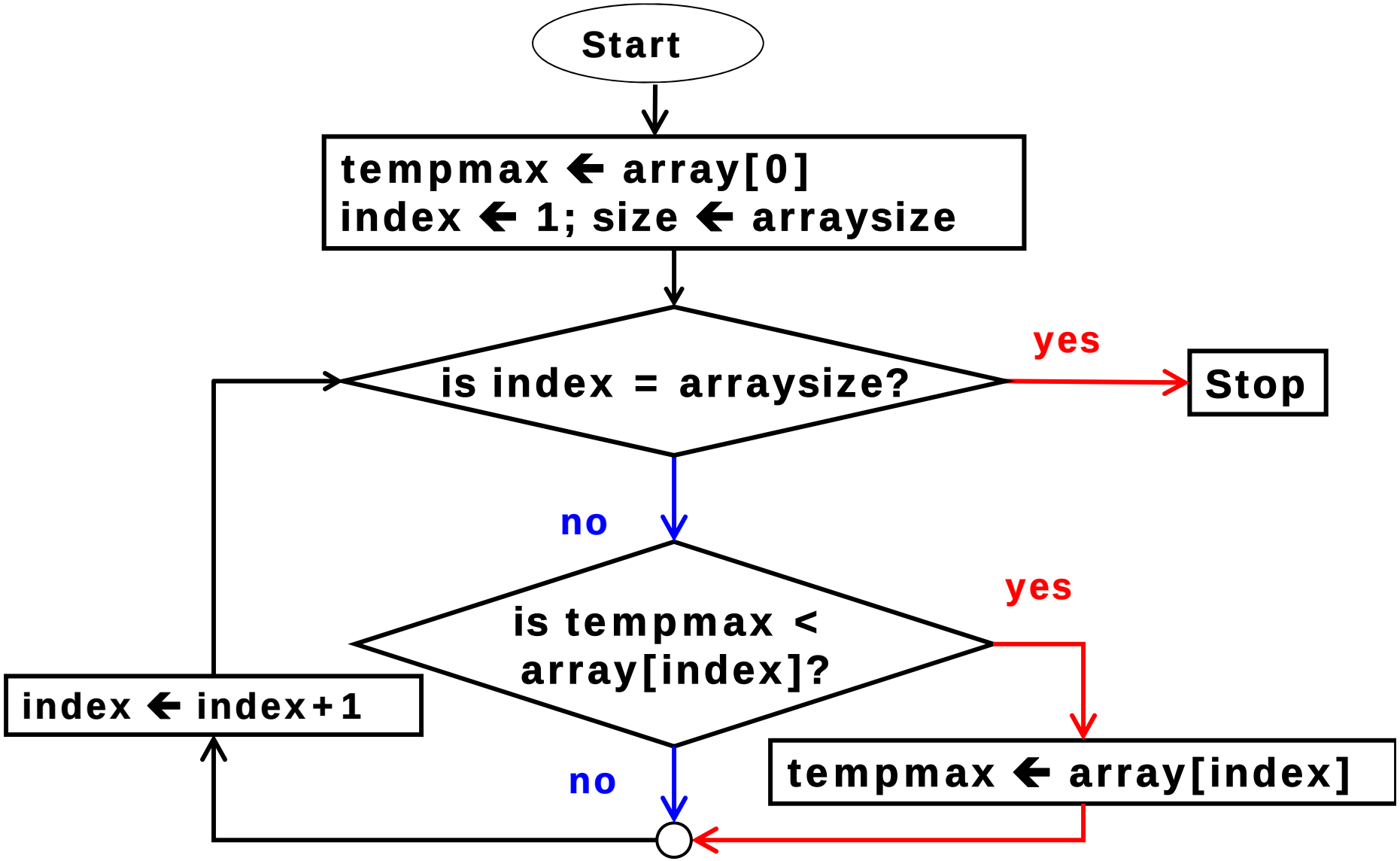
Input parameters: address of array and size of array

Pseudocode

```
tempmax = array[o];  index = 1

while (index < arraysize) do
    if tempmax < array[index]
            then tempmax = array[index]
    index = index + 1
end while

return tempmax as solution
```

Flowchart: finding the maximum element in a 1 dimensional array

Assumption: array is not empty (size > = 1)



Code: finding the maximum element in a 1 dimensional array

```
@*** Max Element in 1 dimensional array  ****
@ Determine the largest integer in an unordered 1D array
@      Assumptions: address of array and size of array as @
       variables and array has at least 1 element
@ ➔ using a few post increments

@ Pseudo- code
@ tempmax = array[0]
@ index = 1
@ while (index < arraysize) do
@      if tempmax < array[index] then tempmax = array[index]
@      index = index + 1
@ end while
@ return tempmax as solution
```

Study by yourself (old Quiz)

```
@ Initialization Phase
@ register usage:
@ r1 <->  array size
@ r2  <-> array address
@ r3  <-> tempmax
@ r0  <-> index count
@ r4  <->   array element
    LDR r1,=size       @get array size
    LDR r1,[r1]
    LDR r2,=array   @ R2 = & array[0]
    LDR r3,[r2],#4  @ r3=tempmax=array[0] and
                    @ r2= & array[1]
    MOV r0,#1   @r0 = index into array
```

```
mainloop:
   CMP r0,r1    @is index = array size?
   BEQ exit     @if true, array finished
   LDR r4,[r2],#4  @else r4=array[i], and
                   @increment pointer
   CMP r3,r4    @compare tempmax to array[i]
   BGE incr     @if tempmax still max, leave it
   MOV r3,r4    @else update tempmax
incr:
   ADD r0,r0,#1     @increment index
   BAL mainloop
exit:
   LDR r2,=max   @store tempmax in variable max
   STR r3,[r2]
   SWI 0x11      @ exit
```

```
@This program determines the largest integer in an unordered non empty array
@Pseudo- code: tempmax = array[0];  index = 1
@while (index < arraysize) do
@  if tempmax < array[index] then tempmax = array[index]
@        index = index + 1; end while; and return tempmax as solution
@ Register use: r1 <->array size; r2<->array address;
@ r3<->tempmax; r0<->index count and r4<-> array element
          .text
          .global _start
          .equ     EXIT,0x11
_start:   LDR      r1,=size          @get array size
          LDR      r1,[r1]
          LDR      r2,=array         @get start of array
          LDR      r3,[r2],#4        @r3=tempmax=array[0] and r2=address of array[1]
          MOV      r0,#1             @r0 = index into array
mainloop: CMP      r0,r1             @is current index = array size?
          BEQ      exit              @if true, array finished
          LDR      r4,[r2],#4        @else get r4=array[i], and increment pointer
          CMP      r3,r4             @compare tempmax to array[i]
          BGE      incr              @if tempmax still max, leave it
          MOV      r3,r4             @else update tempmax
incr:     ADD      r0,r0,#1          @increment index
          BAL      mainloop
exit:     LDR      r2,=max           @store tempmax in variable max
          STR      r3,[r2]
          SWI      EXIT
          .data
size:     .word    10
array:    .word    10,-2,12, 13,-5,6,9,11,13,-2
max:      .skip    4
          .end
```

Good to practice code tracing

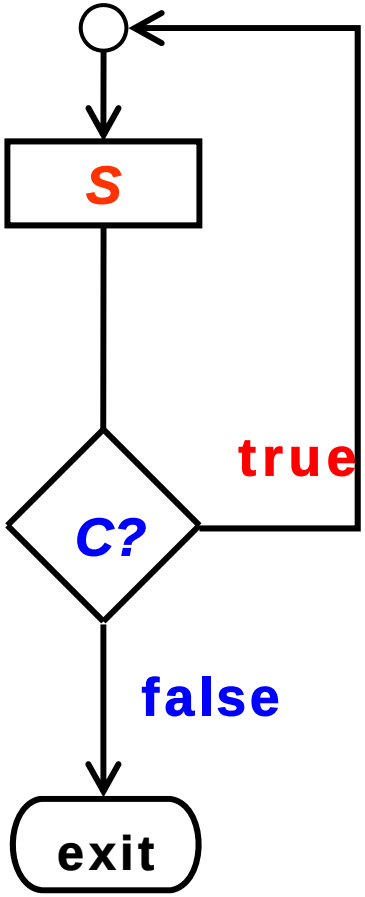
```
@*** DIVISION BY REPEATED SUBTRACTIONS ***

@ Given the numbers M >= 0 and N >0, this program implements
@ "M DIV N = quotient" and
@ "M MOD N = remainder" using repeated subtractions

@ Algo: Repeatedly subtract the divisor N from M, as in M = M - N.
@ Count the number of iterations Q until M < 0.
@ This is one too many iterations and the quotient is Q - 1.
@ The remainder is M + N, the last positive value of M.

@ Pseudo-code 1
@       integers: m, n, quotient, remainder
@       quotient = 0
@       DO
@               quotient = quotient + 1
@               m = m - n
@       WHILE m >= 0
@
@       quotient = quotient - 1
@       remainder = m + n
```

GOOD DOCUMENTATION!

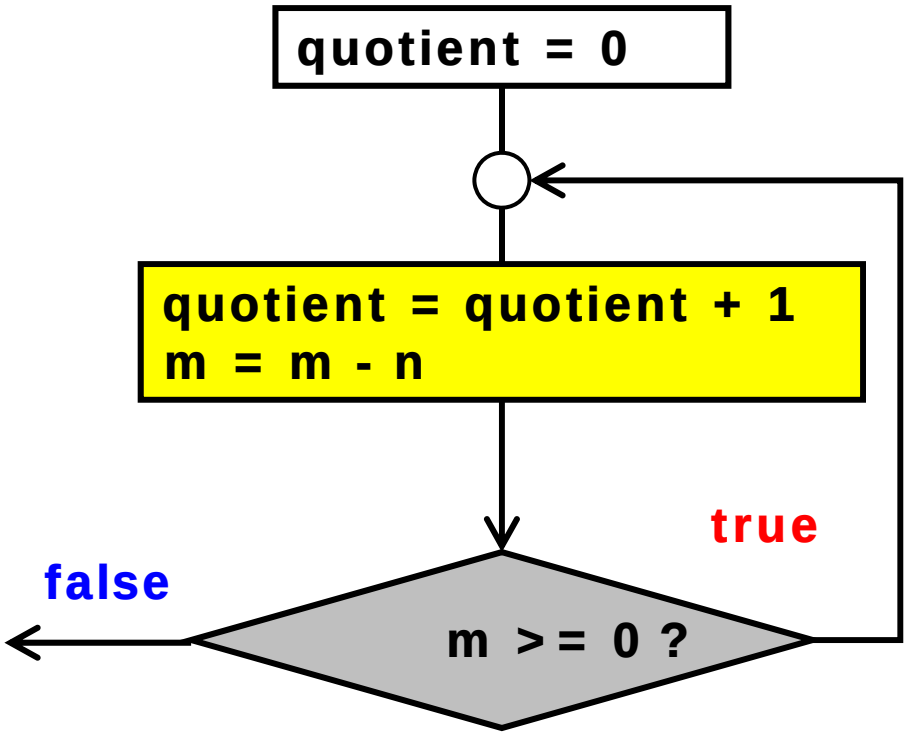
```
@ ========== Data ==========

        .data           @ Begin the "data" segment
        .align          @ Ensure next item begins at a word
                        @(aligned)  address
m:      .word 7
n:      .word 3
quot:   .word 0
rem:    .word 0

        .end
```

REPEAT:

DO *S* WHILE *C*



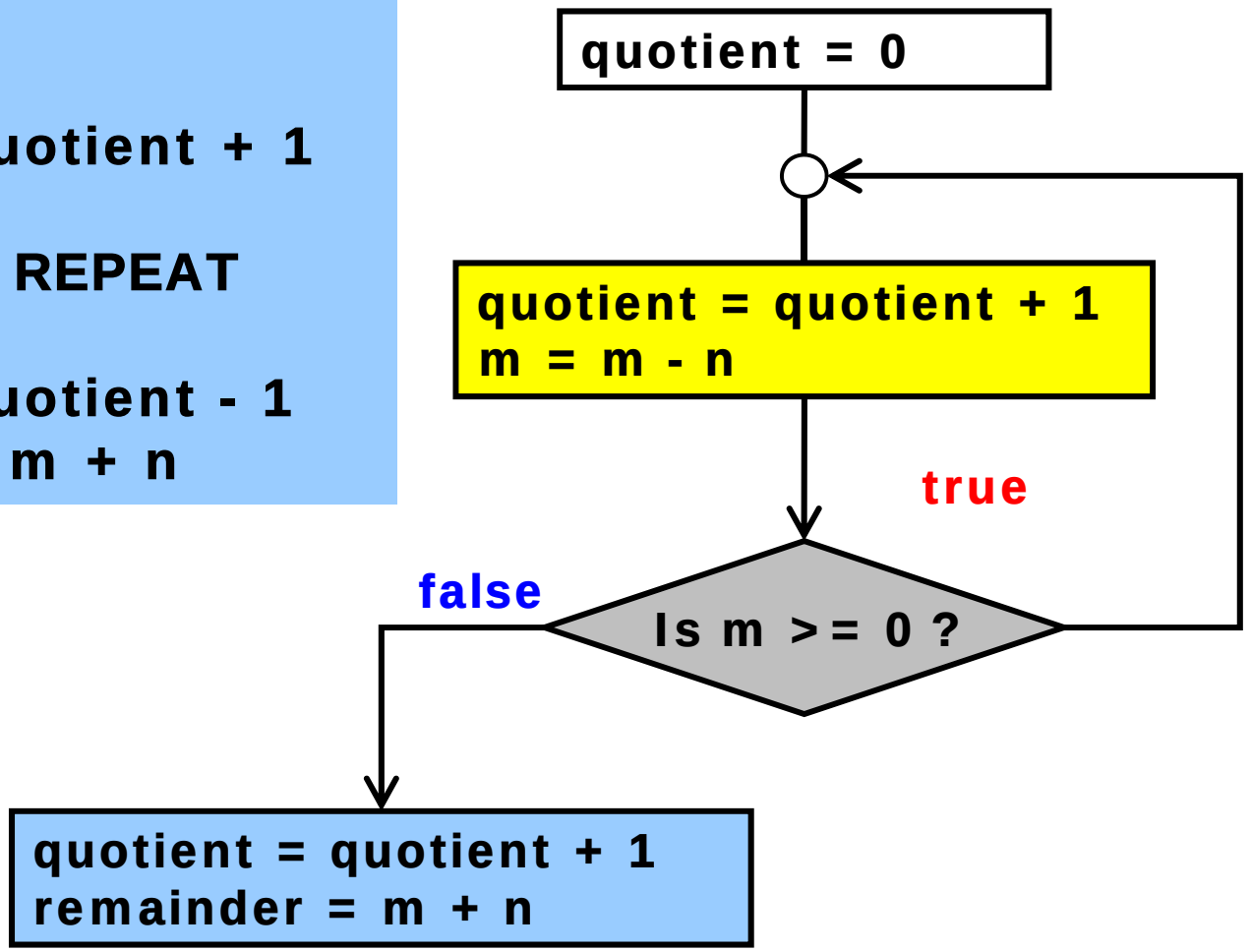

```
@       quotient = 0
@       DO
@               quotient = quotient + 1
@               m = m - n
@       WHILE m >= 0
@
@       quotient = quotient - 1
@       remainder = m + n
```



```
@       quotient = 0
@       DO
@               quotient = quotient + 1
@               m = m - n
@       WHILE m >= 0
@
@       quotient = quotient - 1
@       remainder = m + n
```

```
@       quotient = 0
@       REPEAT
@               quotient = quotient + 1
@               m = m - n
@       IF m >= 0 go to REPEAT
@       ELSE
@               quotient = quotient - 1
@               remainder = m + n
```

```
quotient = 0
REPEAT
    quotient = quotient + 1
    m = m - n
IF m >= 0 go to REPEAT
ELSE
    quotient = quotient - 1
    remainder = m + n
```



Follow all the steps before coding!

```
@ ========= Text (Code) =========
    .text
    .global  _start
_start:
    ldr    r1,=m          @r1 = address of m (dividend)
    ldr    r1,[r1]        @r1 = value of m
    ldr    r2,=n          @r2 = address of n (divisor)
    ldr    r2,[r2]        @r2 = value of n
    mov    r3,#0          @r3 = quotient = 0
loop:
    add    r3,r3,#1       @r3 = quotient + 1
    subs   r1,r1,r2       @compute m - n
    bpl    loop
setresult:
    sub    r3,r3,#1       @r3 = correct quotient
    ldr    r4,=quot       @r4 = address of quotient
    str    r3,[r4]        @store quotient
    add    r1,r1,r2       @r1 = remainder
    ldr    r4,=rem        @r4 = address of remainder
    str    r1,[r4]        @store remainder
    swi    0x11
```