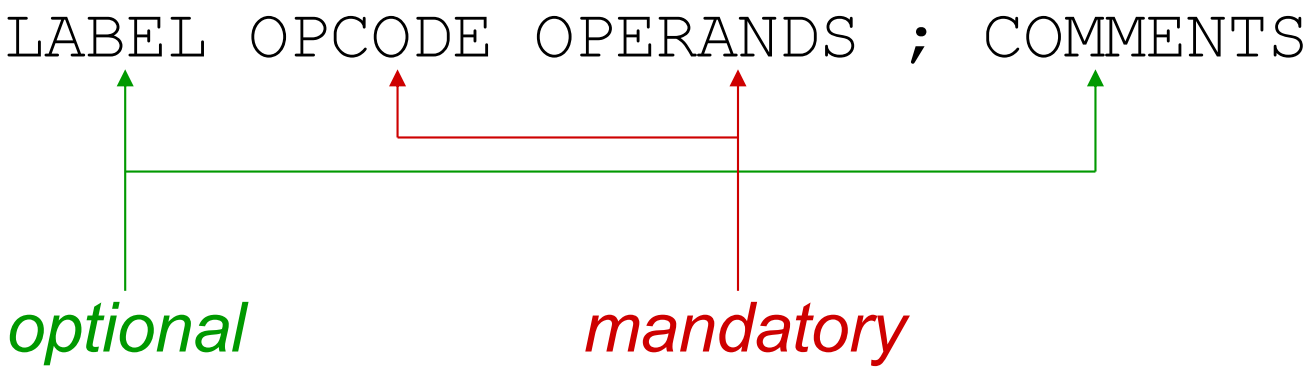
Programming in LC3 Assembly Language (Chapter 6-10)

LC-3 Assembly Language Syntax

- Each line of a program is one of the following:
  - an instruction
  - an assember directive (or pseudo-op)
  - a comment
- Whitespace (between symbols) and case are ignored.
- Comments (beginning with ";") are also ignored.

- An instruction has the following format:



Opcodes and Operands

- Opcodes
  - reserved symbols that correspond to actual (LC-3) instructions
  - listed in Appendix A
    - ex: `ADD`, `AND`, `LD`, `LDR`, ...
- Operands
  - registers -- specified by Rn, where n is the register number
  - numbers -- indicated by # (decimal) or x (hex)
  - label -- symbolic name of memory location
  - separated by comma
  - number, order, and type correspond to instruction format
    - ex:

```
ADD     R1,R1,R3
ADD     R1,R1,#3
LD      R6,NUMBER
BRz     LOOP
```

Labels and Comments

- Label
  - placed at the beginning of the line
  - assigns a symbolic name to the address corresponding to line
    - ex:

```
LOOP    ADD     R1,R1,#-1
        BRp     LOOP
```

- Comment
  - anything after a semicolon is a comment
  - ignored by assembler
  - used by humans to document/understand programs
  - tips for useful comments:
    - avoid restating the obvious, as “decrement R1”
    - provide additional insight, as in “accumulate product in R6”
    - use comments to separate pieces of program

Assembler Directives

- Pseudo-operations.. To make programmer's life easier
  - do not refer to operations executed by program
  - used by assembler
  - look like instruction, but "opcode" starts with dot

Opcode	*Operand*	*Meaning*
`.ORIG`	address	starting address of program
`.END`		end of program
`.BLKW`	n	allocate n words of storage
`.FILL`	n	allocate one word, initialize with value n
`.STRINGZ`	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Trap Codes

- LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them… *more on TRAP instructions later*…

Code	*Equivalent*	*Description*
HALT	**TRAP x25**	**Halt execution and print message to console.**
IN	**TRAP x23**	**Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].**
OUT	**TRAP x21**	**Write one character (in R0[7:0]) to console.**
GETC	**TRAP x20**	**Read one character from keyboard. Character stored in R0[7:0].**
PUTS	**TRAP x22**	**Write null-terminated string to console. Address of string is in R0.**

```
; Example Assembly Program - Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
        .ORIG x3000  ;program starts at address x3000
        LD     R1, PLACE1    ; PLACE is location in memory
               ; note: offset not specified by programmer
        BRn Done  ;if number is Negative goto end
        ADD    R3, R1, #2    ; Add 2 store into R3
        ST     R3, PLACE2    ; store result into PLACE2
Done    HALT   ;halt program
;
PLACE2 .BLKW  1 ; reserve/set aside one word in memory
PLACE1 .FILL  x0005  ; initialize number to 5

        .END   ; end of program
```

```
; Example Assembly Program – Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
        .ORIG x3000   ;program starts at address x3000
        LD      R1, PLACE1     ; PLACE is location in memory
                ; note: offset not specified by programmer
        BRn Done   ;if number is Negative goto end
        ADD     R3, R1, #2     ; Add 2 store into R3
        ST      R3, PLACE2   ; store result into PLACE2
Done    HALT    ;halt program
;
PLACE2 .BLKW  1
PLACE1 .FILL  x0005

        .END    ; end of program
```

Must have Opcode and Operands

Label

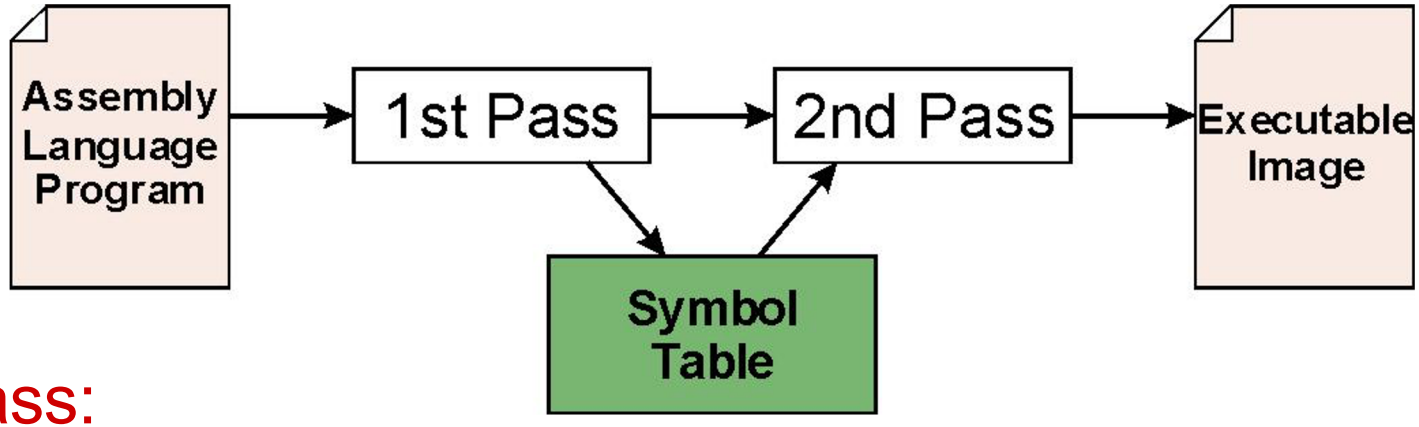
```
; Example Assembly Program – Add 2 to non-negative
number and store into another memory location
; load number from locations PLACE1,
        .ORIG x3000   ;program starts at address x3000
        LD      R1, PLACE1     ; PLACE is location in memory
                ; note: offset not specified by programmer
        BRn Done   ;if number is Negative goto end
        ADD     R3, R1, #2     ; Add 2 store into R3
        ST      R3, PLACE2   ; store result into PLACE2
Done    HALT    ;halt program
;
PLACE2  BLKW  1
PLACE1 .FILL  x0005

        .END    ; end of program
```

Decimal	#
Binary	b
Hex	x

.BLKW Assembler Directive (reserve one location with label 'PLACE2')
.FILL Assembler Directive- reserve one location with label PLACE1 and initialize to 5

Assembly Process

- Assembler: Converts assembly language file (.asm) into an executable file (.obj) …*for the LC-3 simulator in our case*.



- First Pass:
  - scan program file
  - find all labels and calculate the corresponding addresses; this is called the *symbol table*
- Second Pass:
  - convert instructions to machine language, using information from symbol table

Programming in assembly..

- Style guidelines
- Problem decomposition and mapping to assembly

Style Guidelines

1. Provide a program header…standard stuff

- Must include a .ORIG directive at start of program and .END at end of program
  - The .ORIG x[address] tells assembler to load the program starting at that address; .ORIG x2000 tells assembler to load the program starting at x2000

2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)
3. Use comments to explain what each register does.
4. Give explanatory comment for most instructions.
5. Use meaningful symbolic names.
   1. Mixed upper and lower case for readability.
   2. ASCIItoBinary, InputRoutine, SaveR1
6. Provide comments between program sections.

Recap: Problem Solving and Problem Decomposition

- With an eye towards writing assembly programming/low-level software

- Flowcharts anyone ?

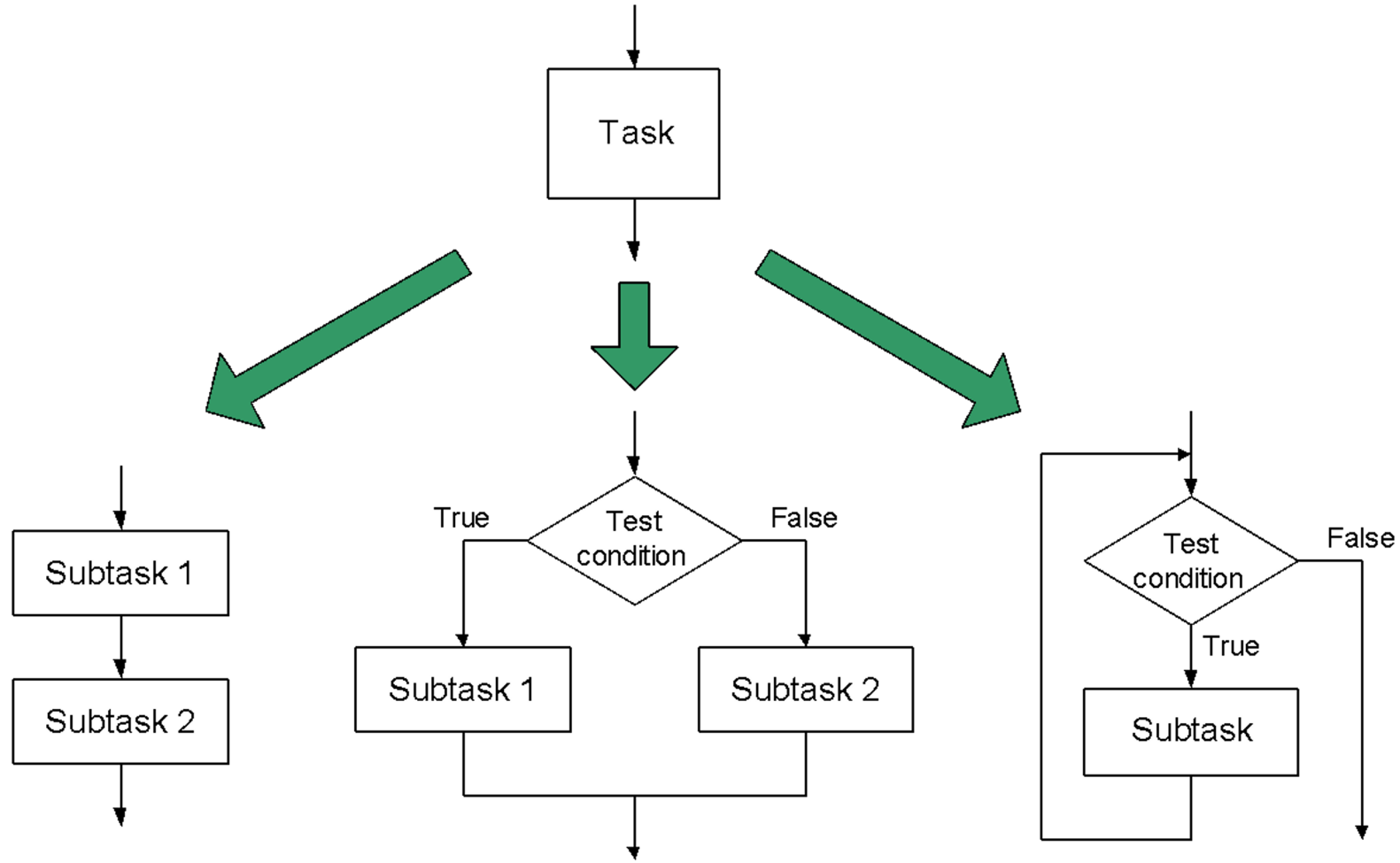
- Decomposition:
  - Break problem/solution into sub-problems/modules
    - Structured programming
  - Connect the modules…
    - With conditionals, iterations, sequence,….

Example

- Array of N numbers, stored starting at address x4000
  - Designate R0 to point to current location in array
  - Initially set R0= x4000
- Read length N of the array and store into register R1
  - For Assume N is 8, therefore initialize R1=8
- Read array element into register R2
- Replace negative numbers by 0
- Add all the (new) numbers and Store/Print the sum
  - Store sum in register R3
  - And store into a memory location x5000
- Example: if numbers starting at x4000 are:
  - 2, -3, 10, 8, -7, 0, 4, -6
- Sum will be 2+10+8+0+4 = 24

Three Basic Constructs

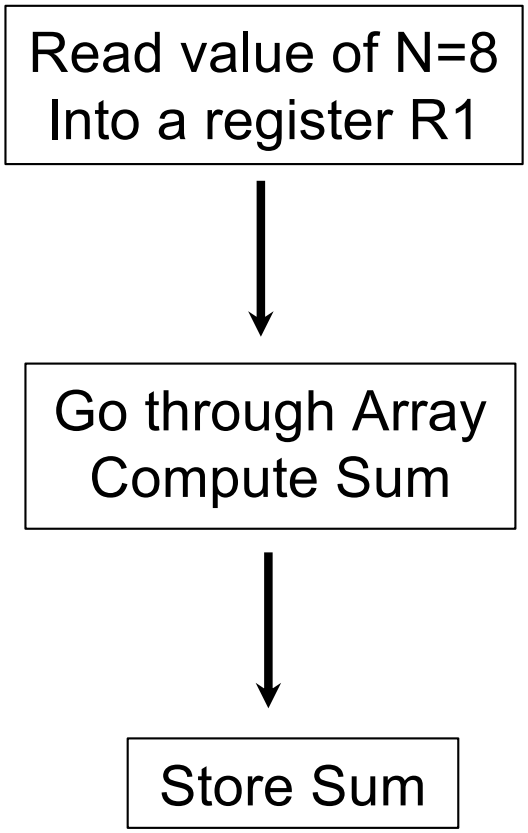
- There are three basic ways to decompose a task:



Sequential

- do Subtask 1, then subtask 2, etc.

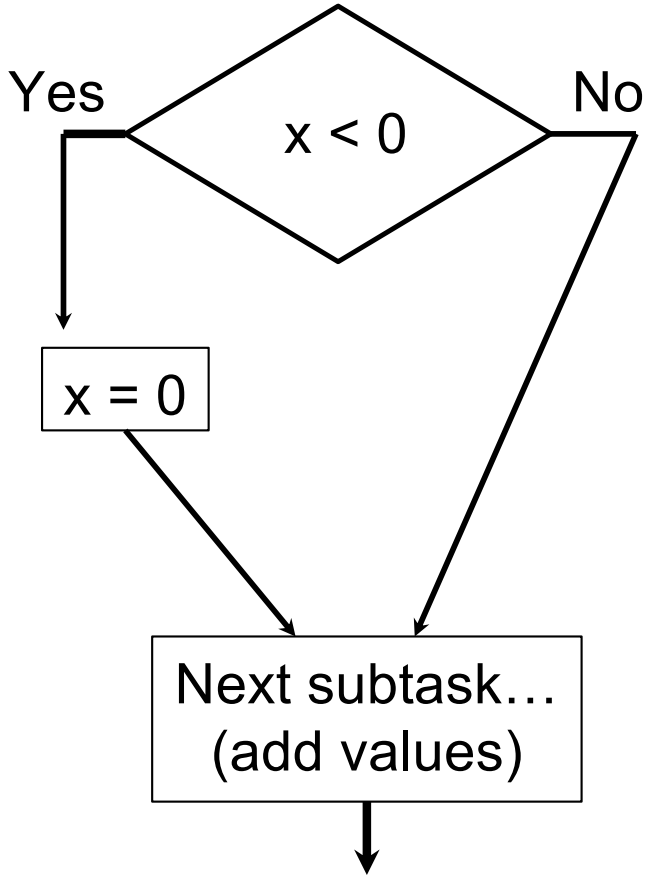
Process Array of Nums
Change –ve to 0 and +ve to +2
and
Compute Sum of numbers
Store Sum



Conditional

- If condition is true, do Subtask 1; else, do Subtask 2.

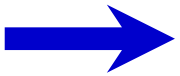
Number is in R2
Check if number >=0
Change –ve to 0

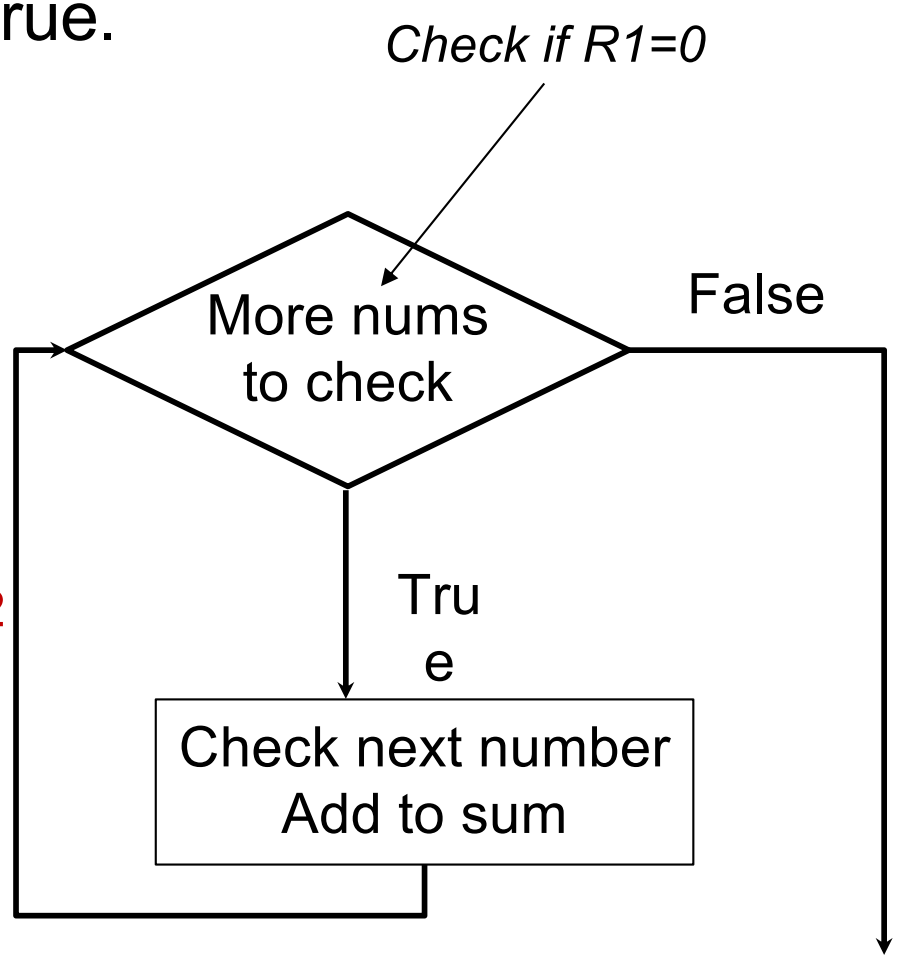


Iterative

- Do Subtask over and over,
as long as the test condition is true.

Check each element
In array and compute sum





Read from array ‘pointer’ R0 into register R2
Check if number is –ve
Increment ‘pointer’ by 1 to point to
next address
Add value to Sum (in register R3)
Decrement R1 by 1 (counts # elements)
Repeat loop

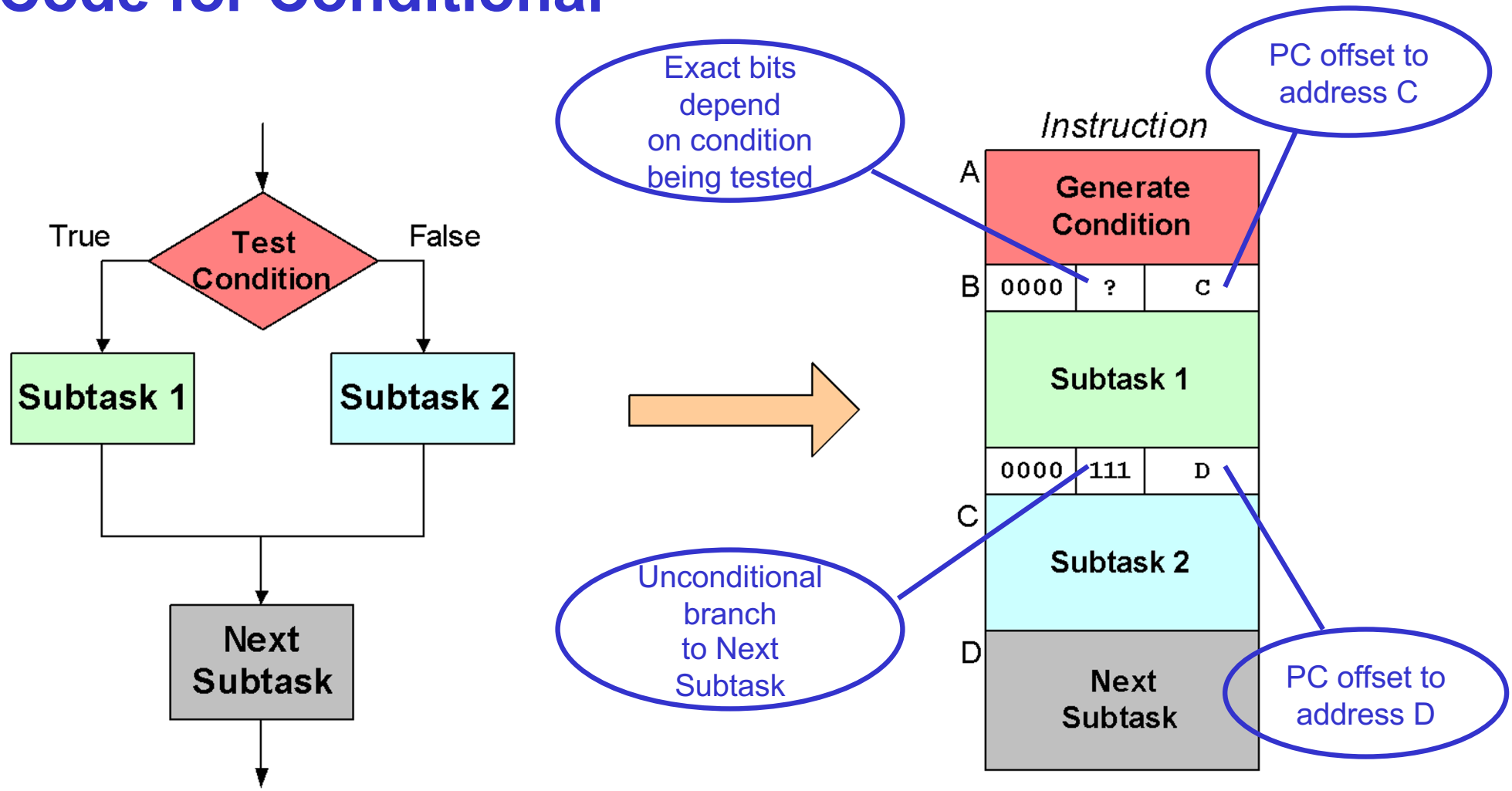
LC-3 Control Instructions

- How do we use LC-3 instructions to encode the three basic constructs?

- Sequential
  - Instructions naturally flow from one to the next, so no special instruction needed to go from one sequential subtask to the next.

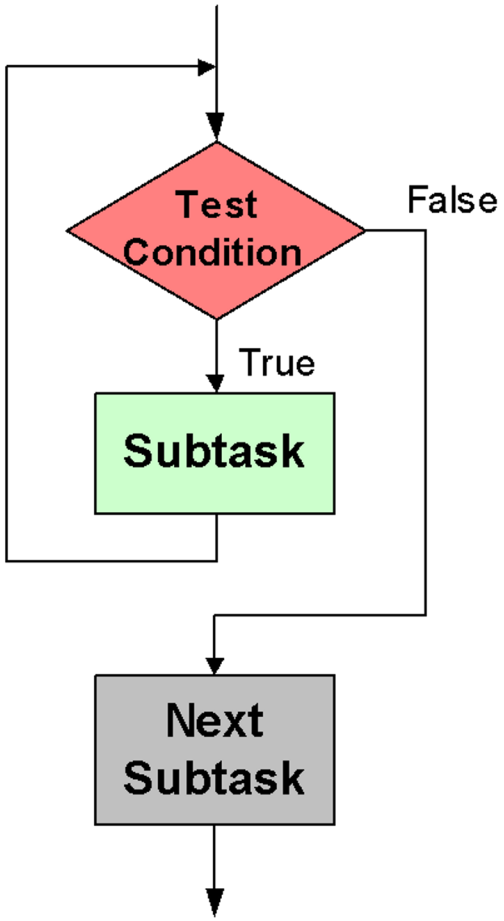
- Conditional and Iterative
  - Create code that converts condition into N, Z, or P.
    Example:
    Condition: “Is R0 = R1?”
    Code: Subtract R1 from R0; if equal, Z bit will be set.
  - Then use BR instruction to transfer control to the proper subtask.
  - Note: BR NZP results in “always branch”

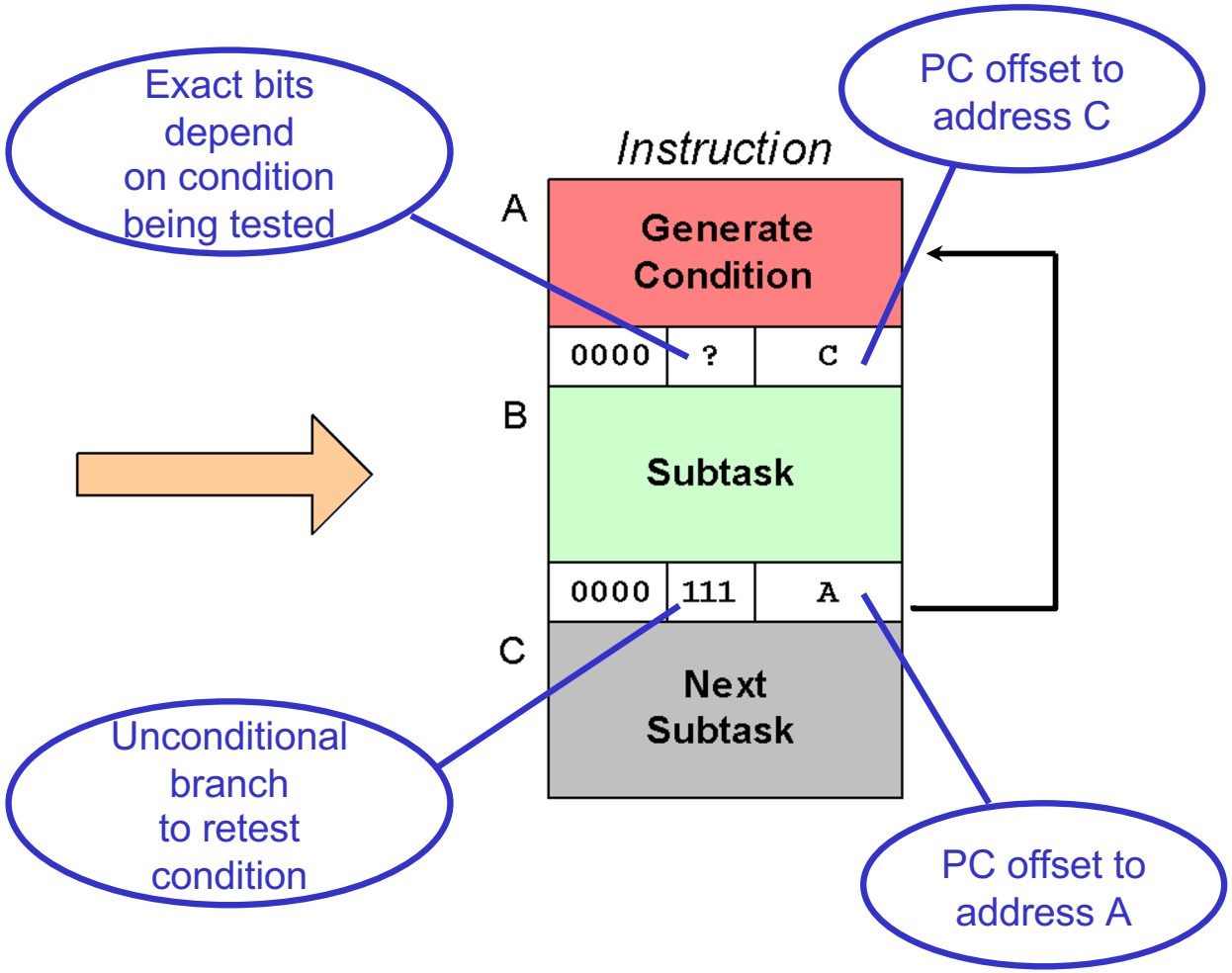
Code for Conditional



Assuming all addresses are close enough that PC-relative branch can be used.

Code for Iteration





Assuming all addresses are on the same page.

Converting Code to Assembly

- Can use a standard template approach
- Typical Constructs
  - if/else
  - while
  - do/while
  - for

if/else

option 1: branch to THEN part

```
if(x > 0) /* load x
into register R1 */
{
      r2 = r3 + r4;
}
else
{
      r5 = r6 + r7;
}
```

```
      LD    R1, X
      BRP   THEN
      ADD R5,R6,R7
      BRNZP DONE
THEN ADD R2,R3,R4
DONE ...

; note: BRNZP is
unconditional branch
Used to go back to start
of loop
```

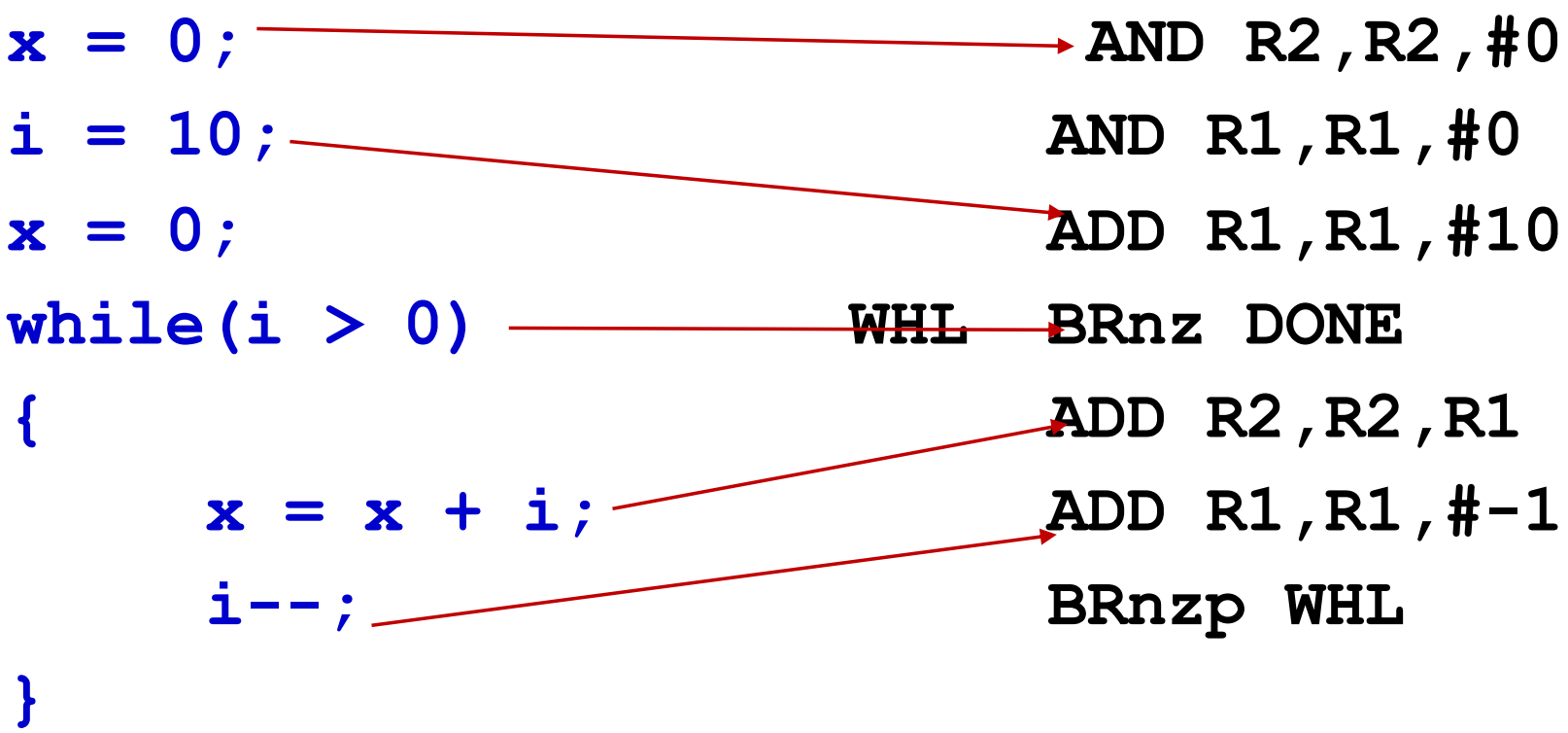
if/else

option 2: branch to ELSE part

```
if(x > 0)
{
     r2 = r3 + r4;
}
else
{
     r5 = r6 + r7;
}
```

```
     LD    R1,X
     BRNZ  ELSE
     ADD R2,R3,R4
     BRNZP DONE
ELSE ADD R5,R6,R7
DONE ...
```

while


x = 0;
i = 10;
x = 0;
while(i > 0)
{
x = x + i;
i--;
}
AND R2,R2,#0
AND R1,R1,#0
ADD R1,R1,#10
WHL BRnz DONE
ADD R2,R2,R1
ADD R1,R1,#-1
BRnzp WHL