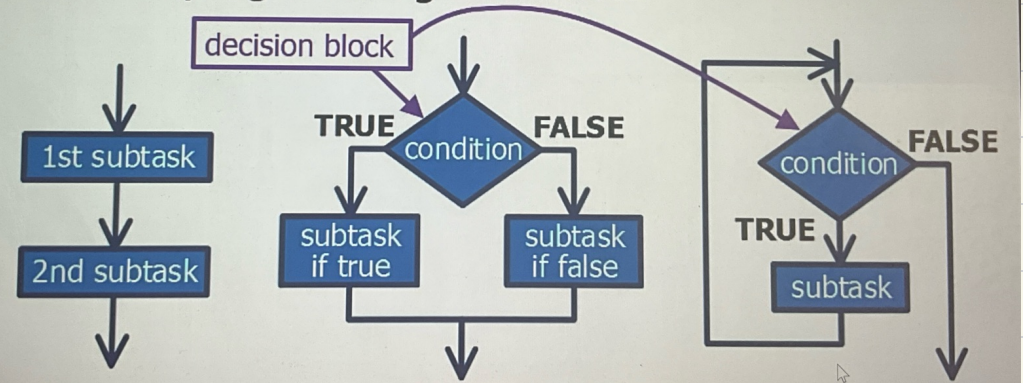
LC-3 Control Flow

Structured Programming

- **Flowcharts** graphically represent programs during their design
  - Writing code for a flowchart is the easy part!
- Three programming constructs:



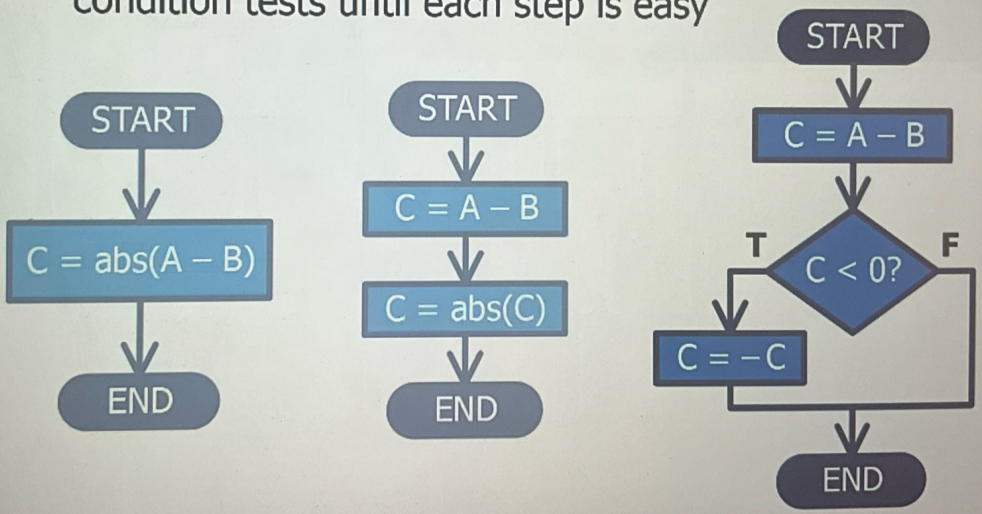
One by one

If/else

Loop

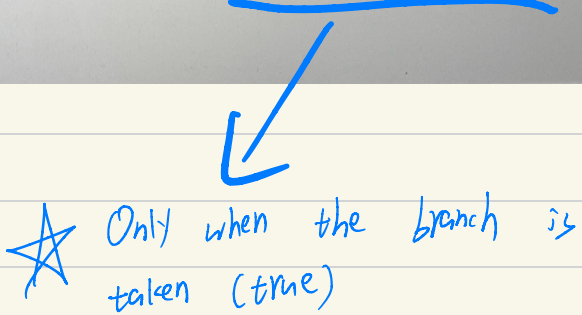
Systematic Decomposition

- Break down complex problems into a set of easier-to-handle tasks
  - Replace each task with set of smaller tasks and condition tests until each step is easy

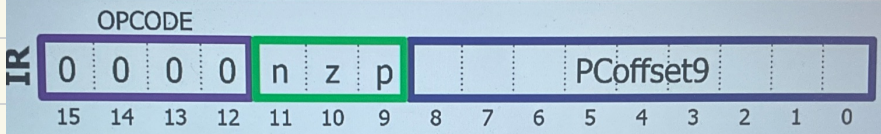


Changing the Flow

- The next instruction to execute does not have to be in the next location in memory
- **Branch** instructions allow the program to change the value in the PC
  - **Unconditional branch**: always updates the PC
  - **Conditional branch**: only updates the PC (**is taken**) if some specified condition is true
    - If the condition is false, the branch **is not taken**, and the next instruction to execute will be the one at the memory location just after the conditional branch instruction
- The new PC value is called the **branch target**



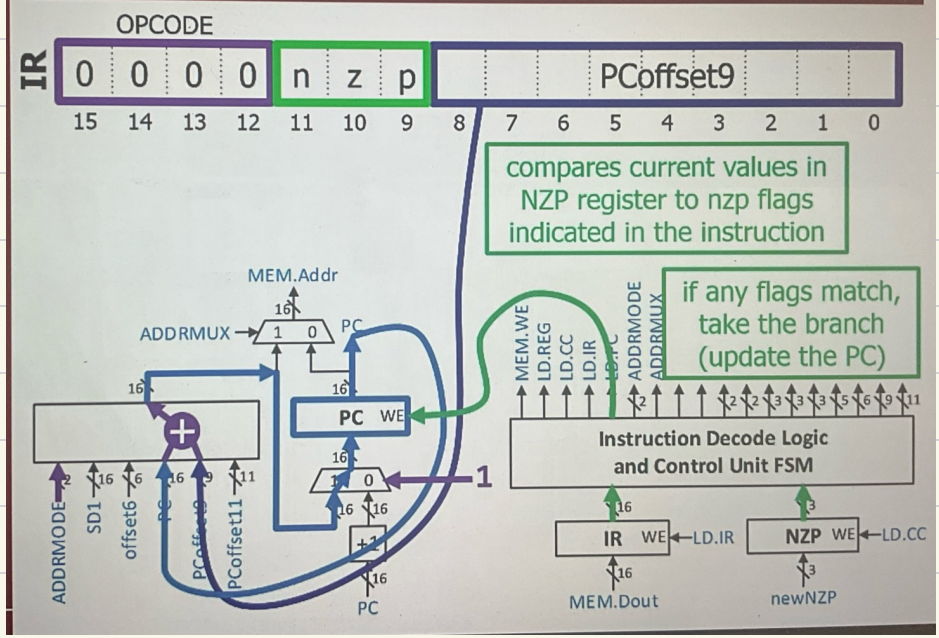
BR Instruction



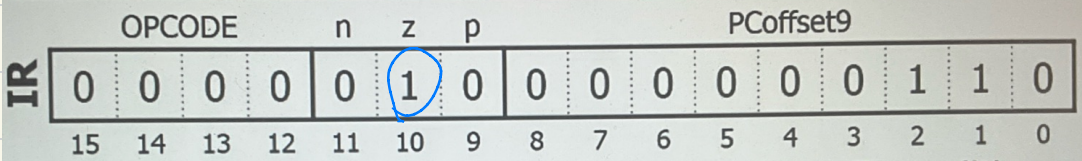
- Calculate the branch target by adding the sign-extended offset to the already-incremented PC
- Compare conditions indicated in the instruction with the current NZP condition code values
  - If any match, then update the PC with the calculated branch target
  - If none match, do not modify the PC

Note: even if the BR does not modify the PC, the PC was incremented when the BR was fetched

BR Instruction

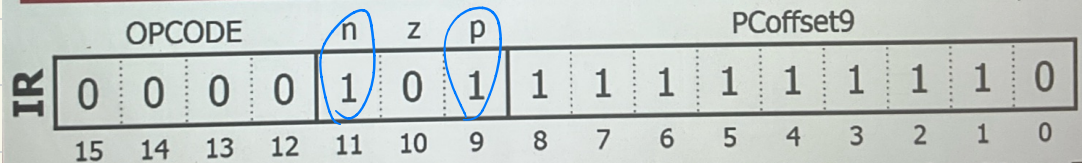


Example BR Instructions



- Add 6 to the already-incremented PC if the **Z** condition code is true

Skip over the next 6 instructions if the most recent result that affected the condition codes was equal to zero



- Add -2 to the already-incremented PC if either the **N** or **P** condition codes are true

Go to the instruction one address before this one if the most recent result affecting the condition codes was non-zero

This will go backward by 2

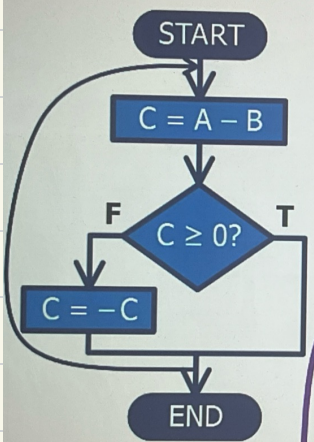
Using BR in LC-3 Assembly Code

- Specify which condition codes should be checked by appending them to the BR opcode mnemonic
  - **BRz** will branch if zero
  - **BRzp** will branch if non-negative
  - **BRnp** will branch if non-zero
  - **BRnzp** will branch unconditionally

can also use **BR** (without flags) for unconditional

- The branch target is specified using a **label** instead of an offset in assembly language
  - BR **START**
  - BRn **IS_NEGATIVE**

Example Program: C ← abs(A–B)



absdiff.asm

```
; Program computes C ← abs(A - B)
; A is R0, B is R1, C is R2
        .ORIG x0200
START
        NOT R1, R1      ; negate B
        ADD R1, R1, #1
        ADD R2, R0, R1  ; A + (-B)
        BRzp SKIPNEG    ; if C>=0, done
        NOT R2, R2      ; C is negative
        ADD R2, R2, #1  ;  so negate it
SKIPNEG
        BR START        ; repeat forever
        .END
```

conditional branch skips negation of C if it is already non-negative

unconditional branch back to START

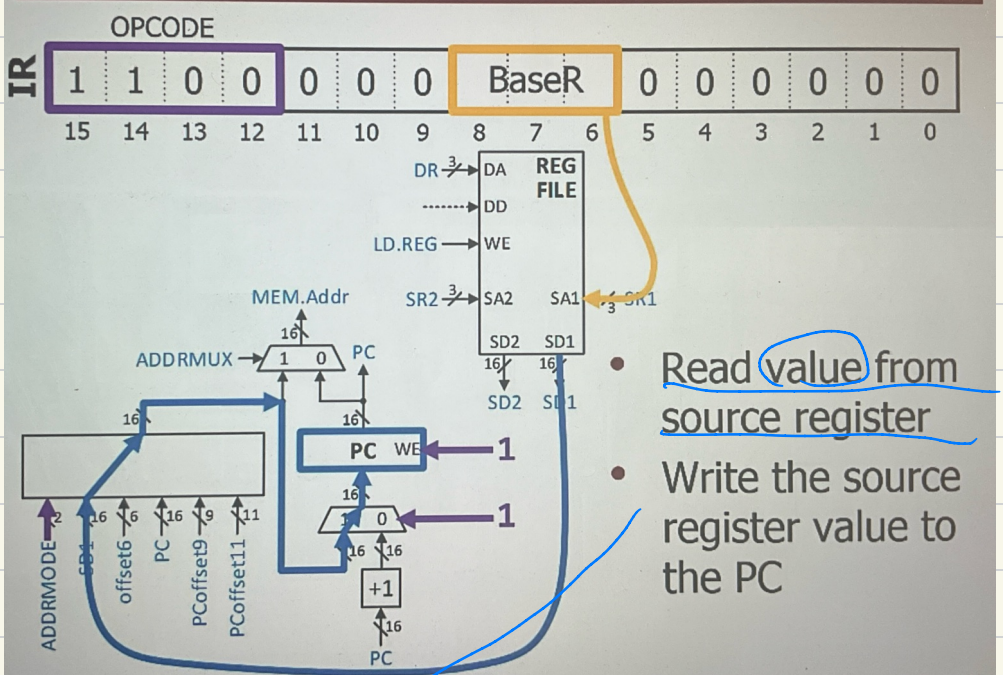
BRzp means if zero or positive, skip to the SKIPNEG label

JMPing Further

- The distance of a BRanch target is limited by the range of offsets encoded in the instruction
  - 9-bit offset, range [-256, 255] from incremented PC
  - If target code is at an address further away in memory, need to use a different instruction...

- JMP is an **unconditional** jump instruction that updates the PC with the value in a register

JMP Instruction



- Read value from source register
- Write the source register value to the PC

The value from the SR is the location for jump

Programming Techniques

Flowchart Design Tips

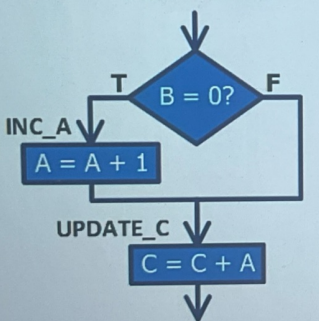
- Solve the problem at a higher level
  - Design the flowchart before writing any code
  - Avoid referring to registers in the flowchart
  - Use variable names that clarify their purpose
- Expect that you'll need to make revisions
  - Use a pencil!
  - Don't pack everything together – leave some room for adding blocks
- Annotate tasks with short but explanatory names
- Test your flowchart before you code

IF: Flowchart

High-Level Language

```
if (B==0) A = A + 1;
C = C + A;
```

Flowchart



- Expresses that a task should be performed only under a certain condition
- **Example:**
  - **If** B is zero, increment A
  - Afterwards, whether B was zero or not, add A to C

Assembly Language

```
; R0 is A, R1 is B, R2 is C

; the AND only needed if conditions
; not already set based on B
    AND  R1, R1, R1 ; test B
    BRz  INC_A      ; taken if B=0
    BRnp UPDATE_C   ; taken if B!=0
INC_A
    ADD  R0, R0, #1 ; increment A

UPDATE_C
    ADD  R2, R2, R0 ; C = C + A
```

B is zero

B is non-zero

NOT Efficient!

Assembly Language

```
; R0 is A, R1 is B, R2 is C

; the AND only needed if conditions
; not already set based on B
    AND  R1, R1, R1 ; test B
    BRz  INC_A      ; taken if B=0
    BRnp UPDATE_C   ; taken if B!=0
INC_A
    ADD  R0, R0, #1

UPDATE_C
    ADD  R2, R2, R0 ; C = C + A
```

B is non-zero

B is zero

program has same behavior without this!

Efficient!

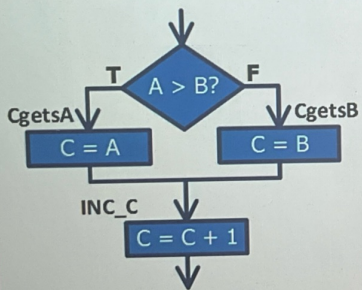
(☆ Branch around, not branch to)

IF...ELSE: Flowchart

High-Level Language

```
if (A > B) C = A;
else C = B;
C = C + 1;
```

Flowchart



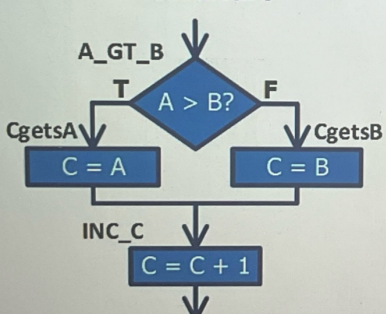
- Expresses that one of two tasks should be performed, based on a given condition
- **Example:**
  - Let C be the max of A, B
  - Afterwards, increment C

IF...ELSE: Initial Implementation

High-Level Language

```
if (A > B) C = A;
else C = B;
C = C + 1;
```

Flowchart



Assembly Language

```
; R0 is A, R1 is B, R2 is C

A_GT_B
   NOT  R4, R1      ; compute (-B)
   ADD  R4, R4, #1
   ADD  R4, R4, R0 ; A + (-B)
   BRp  CgetsA      ; taken if A > B
   BRnz CgetsB      ; taken if A ≤ B
CgetsB
   ADD  R2, R1, #0 ; C = B
   BR   INC_C
CgetsA
   ADD  R2, R0, #0 ; C = A
   BR   INC_C
INC_C
   ADD  R2, R2, #1 ; increment C
```

program has same behavior without these!

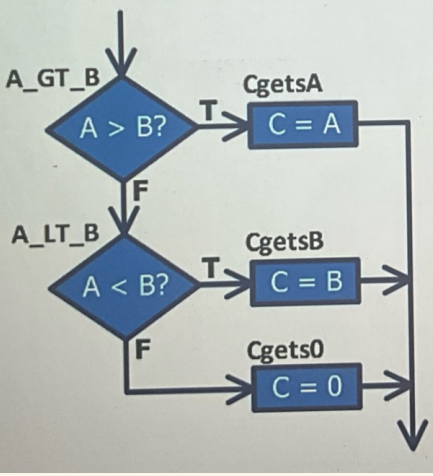
(☆ Do not need branch for Every arrow!)

IF...ELSE IF: Flowchart/Code

High-Level Language

```
if (A > B) C = A;
else if (A < B) C = B;
else C = 0;
```

Flowchart



Assembly Language

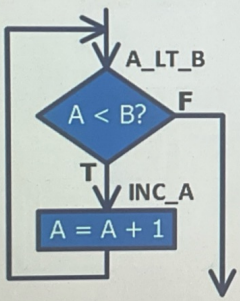
```
; R0 is A, R1 is B, R2 is C
A_GT_B
    NOT  R4, R1      ; compute (-B)
    ADD  R4, R4, #1
    ADD  R4, R4, R0 ; A + (-B)
    BRp  CgetsA      ; taken if A > B
A_LT_B
    BRn  CgetsB      ; taken if A < B
Cgets0               ; executed if A=B
    AND  R4, R4, #0 ; C = 0
    BR   REST_OF_CODE
CgetsB
    ADD  R2, R1, #0 ; C = B
    BR   REST_OF_CODE
CgetsA
    ADD  R2, R0, #0 ; C = A
; executed after if...else if...else
REST_OF_CODE
; remaining program code
```

WHILE: Flowchart

High-Level Language

```
while (A < B) A = A + 1;
```

Flowchart



- Expresses that, as long as a condition remains true, a task should be repeated
- **Example:**
  - **While** A is less than B, increment A

Assembly Language

```
; R0 is A, R1 is B

A_LT_B
    NOT   R4, R1          ; compute (-B)
    ADD   R4, R4, #1
    ADD   R4, R4, R0      ; A + (-B)
    BRzp  REST_OF_CODE    ; taken if A ≥ B

INC_A                     ; exec. if A < B
    ADD   R0, R0, #1      ; increment A
    BR    A_LT_B

; executed after while loop
REST_OF_CODE
; remaining program code
```

Assembly Language

```
; R0 is A, R1 is B, R4 is (-B)
NEG_B                     No need to recompute every time
    NOT   R4, R1          ; compute (-B)
    ADD   R4, R4, #1
A_LT_B
    ADD   R5, R4, R0      ; A + (-B)
    BRzp  REST_OF_CODE    ; taken if A ≥ B

INC_A                     ; exec. if A < B
    ADD   R0, R0, #1      ; increment A
    BR    A_LT_B

; executed after while loop
REST_OF_CODE
; remaining program code
```

Outside the loop

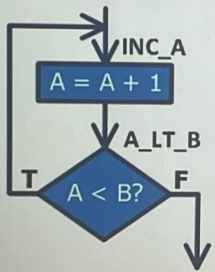
Change the DR so that 124 stays equal to -B in the loop!

DO...WHILE: Flowchart

High-Level Language

```
do (A = A + 1)
while (A < B);
```

Flowchart



- Repeat a task as long as a condition remains true
  - Similar to the WHILE construct, but the task is performed at least once
- **Example:**
  - Increment A until A is no longer less than B
  - Or... **do** an increment of A **while** A is less than B

Assembly Language

```
; R0 is A, R1 is B, R4 is (-B)

NEG_B
    NOT  R4, R1        ; compute (-B)
    ADD  R4, R4, #1

INC_A
    ADD  R0, R0, #1    ; increment A

A_LT_B
    ADD  R5, R4, R0    ; A + (-B)
    BRn  INC_A         ; taken if A < B

; executed after do...while loop
REST_OF_CODE
; remaining program code
```

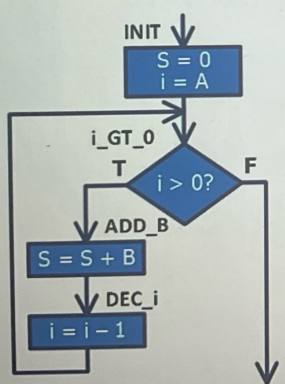
Different order than WHILE loop

FOR: Flowchart

High-Level Language

```
S = 0;
for (i = A; i > 0; i--)
   S = S + B;
```

Flowchart



- Expresses a loop that should be repeated a certain number of times
- **Example:**
  - Multiplication (A × B) using repeated addition
    - Initialize S to 0, then add B to the sum A times

Assembly Language

```
; R0 is A, R1 is B, R2 is S, R3 is i

INIT
   AND  R2, R2, #0   ; clear S
   ADD  R3, R0, #0   ; init loop count

i_GT_0
   BRnz REST_OF_CODE ; taken if i ≤ 0

ADD_B                ; exec. if i > 0
   ADD  R2, R2, R1   ; S = S + B
DEC_i
   ADD  R3, R3, #-1  ; decr loop count
   BRp  ADD_B        ; loop if i>0

; executed after for loop
REST_OF_CODE
; remaining program code
```

Common Coding Pitfalls

- Inefficient branches
  - Branching to the next instruction
  - Unnecessary branches around other branches

```
   BR   INC_C
INC_C
```

```
   BRz  INC_A
   BRnp UPDATE_C
INC_A
   ADD  R0, R0, #1
UPDATE_C
   ADD  R2, R2, R0
```

- Using fewer conditions when a branch should be unconditional

```
START
    ; some code
ITERATE
    ; some code
   BRnz ITERATE
   BRp START
```

- Using labels that have no meaning

```
LABEL1
   ; some code
LABEL2
```

Avoid Other Common Pitfalls

- Initialization
  - DO initialize registers that need to be initialized
  - DO NOT clear a register unless it needs to be cleared

```
; R0                                   R3 is i
INIT
   AND  R2, R2, #0   ; clear S
   ADD  R3, R0, #0   ; init loop count
i_GT
   BR                ; if i ≤ 0

ADD_B                ; exec. if i > 0
   ADD  R2, R2, R1   ; S = S + B
DEC_i
   ADD  R3, R3, #-1  ; decr loop count
   BRp  ADD_B        ; taken if i > 0

; executed after for loop
REST_OF_CODE
; remaining program code goes here
```

R2 needs to be cleared (cannot assume it is 0)

R0 can be copied to R3 without clearing R3 first