# Review: LC-3 ISA and assembly language programming

## Lecture Topics

- Course overview
- LC-3 ISA review
- LC-3 assembly language review
- Programming example

## Reading assignments

- Course Overview and Policies on the course's wiki
- Textbook Ch. 1-5

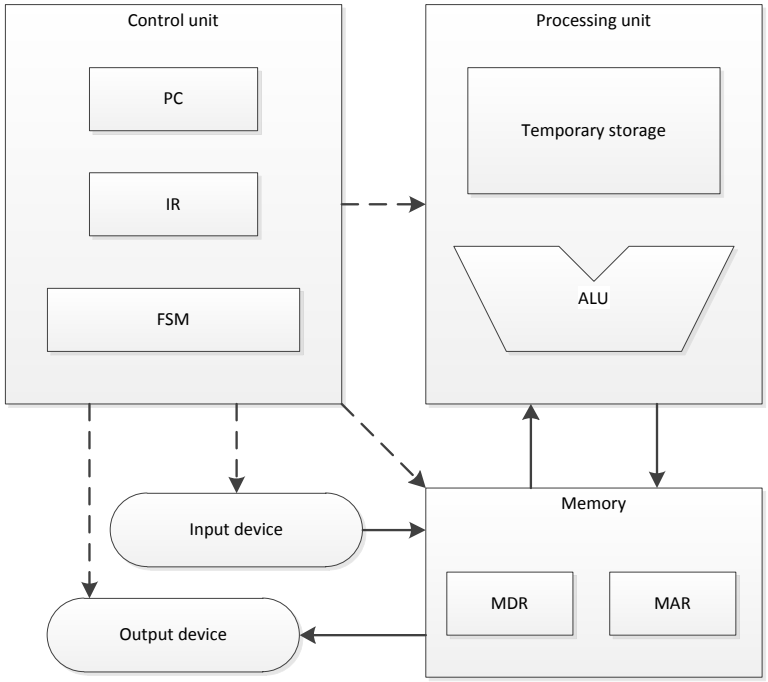
# Course overview and orientation

- Read Course Overview and Policies at https://wiki.cites.illinois.edu/wiki/display/ece220/Home

# LC-3 ISA review

## Von Neumann model components

- Memory – stores data and program
- Processing unit – performs the data processing
- Input – means to enter data and program
- Output – means to extract results
- Control unit – controls the order of the instruction execution



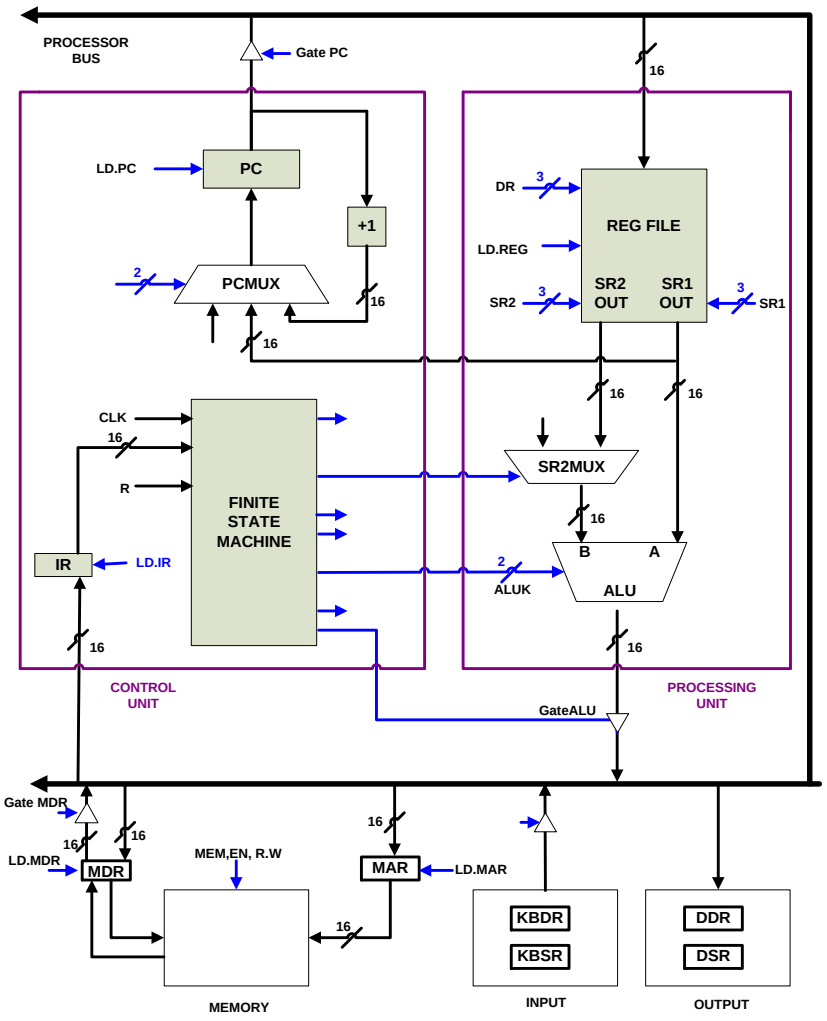

## LC-3 as a von Neumann machine

- In this course we will use a computer, called Little Machine 3, or LC-3 which is an implementation of the basic von Neumann architecture.

### LC-3 data path

- The LC-3 data path consist of all the components that process the information during an instruction cycle
  - The functional unit that operates on the information
  - The registers that store the information
  - Buses and wires that carry information from one point to another
- **Basic components of the data path:**
- The global bus
  - LC-3 global bus consists of 16 wires and associated electronics

  - It allows one structure to transfer up to 16 bits of information to another structure by making the necessary electronic connections on the bus
  - Exactly one value can be translated on the bus at a time
  - Each structure that supplies values to the bus connects to it via a *tri-state device* that allows the computer's control logic to enable exactly one supplier of information to the bus at a time
  - The structure wishing to obtain information from the bus can do so by asserting (setting to 1) its LD.x (load enable) signal.
- Memory
  - Memory in LC-3 is accessed by loading the memory address value into MAR
  - The result of memory read is stored in MDR
  - The computer's control logic supplies the necessary control signals to enable the read/write of memory
- The ALU and the register file
  - The ALU is the processing element
    - Has two inputs and one output
    - Inputs come directly from the register file
    - One of the inputs can also be supplied directly from the IR, controlled by the SR2MUX
    - Output goes to the bus
      - It then is stored in the register file and
      - Processed by additional circuitry to generate the condition codes N,Z,P
  - The register file consists of 8 16-bit registers
    - It can supply up to two values via its two output ports
    - It can store one value, coming from the bus
    - Read/write access to the register file is controlled by the 3-bit resister signals, DR, SR1, SR2
- The PC and PCMUX
  - The PC supplies via the global bus to the MAR the address of the instruction to be fetched at the start of the instruction cycle. The PC itself is supplied via the three-to-one PCMUX, depending on the instruction being executed
  - During the fetch cycle, the PC is incremented by 1 and written to the PC register
  - If the executed instruction is a control instruction, the relevant source of the PCMUX depends on the type of the control instruction and is computed using a special ADD unit
- The MARMUX
  - Controls which of two sources will supply memory address to MAR
  - It is either the value coming from the instruction register needed to implement TRAP instruction, or the value computed based on the PC or a value stored in one of the general-purpose registers

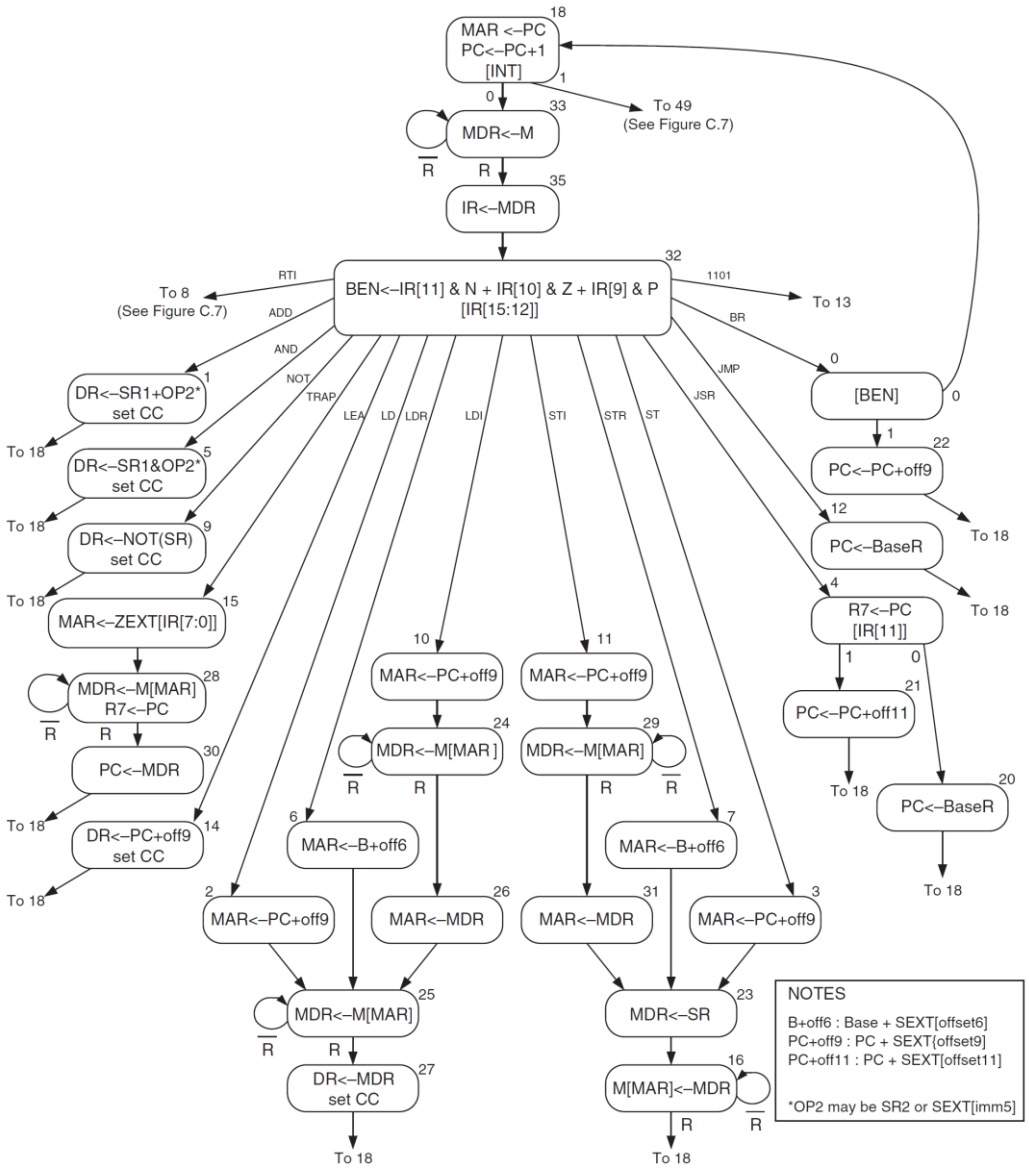



## LC-3 instruction cycle

- FETC
  - IR contains the fetched instruction
  - PC is incremented by 1
- DECODE
  - Instruction is decoded resulting in control logic providing various control signals to the computer
- EVALUATE ADDRESS
  - Address of memory to be accessed during the execution of the instruction is computed
  - Memory access instructions require this phase
- OPERAND FETCH
  - The data from memory is loaded into MDR
- EXECUTE
  - ALU is directed to perform the operation
  - Memory access instructions do not have this phase
- STORE RESULTS
  - Results are stored to memory or registers, depending on the instruction

## LC-3 FSM

- State machine consists of 52 distinct states
  - Shown on page 568 (C.2) of the textbook

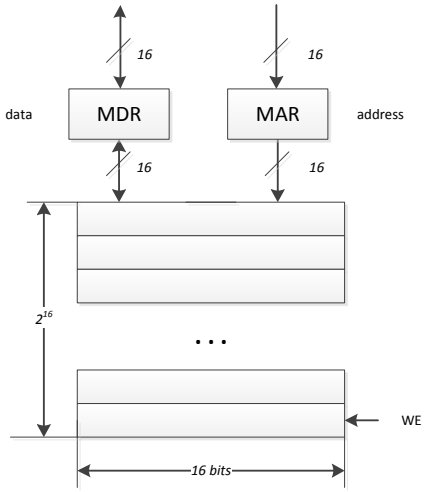



## LC-3 ISA

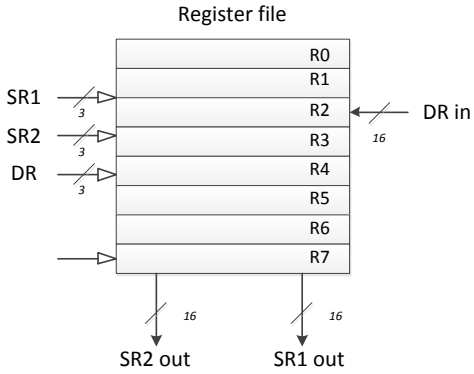
- ISA specifies all the information about the computer that the software has to be aware of
- ISA defines an interface for the programmer that he needs to know when writing programs in the computer's own machine language
- ISA specifies the following
  - Memory organization
  - Resister set
  - Instruction set
  - Data types
  - Addressing models

## *LC-3 ISA overview*

- Memory organization
  - 16-bit addressable
  - $2^{16}$ address space




- Register set
  - 8 16-bit general-purpose registers, named R0-R7

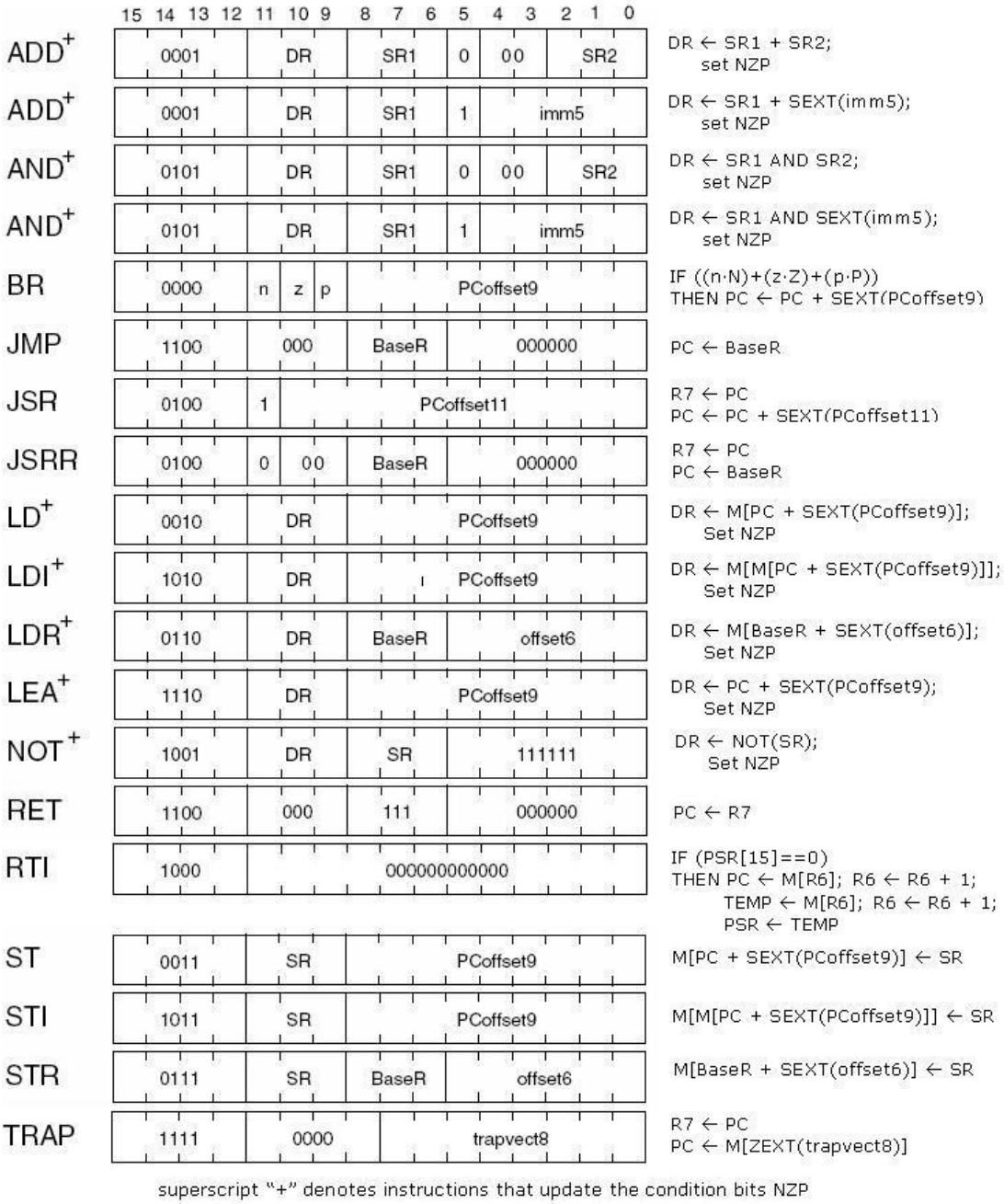



  - Other special-purpose registers, such as PC and IR
- Data types
  - 16-bit 2's complement integers
- Addressing models
  - A mechanism for specifying where the operands are located
  - Non-memory addresses
    - immediate (part of the instruction)
    - register
  - Memory addresses
    - PC-relative, base+offset, indirect
- The instruction set
  - 16-bit instructions
  - Bits 12-15 of the 16-bit instruction are used to specify the opcode
  - Operate instructions: ADD, AND, NOT

- Data movement instructions: LD, LDI, LDR, LEA, ST, STR, STI
- Control instructions: BR, JSR/JSSR, JMP, RET, TRAP, RTI

- Condition codes
  - LC-3 has 3 single-bit registers that are set to 0 or 1 each time one of the general-purpose registers (R0-R7) is **written to**
    - N, Z, P – condition codes
    - N=1 (Z=P=0) when the value stored in one of the registers is negative
    - P=1 (Z=N=0) when the value stored in one of the registers is positive
    - Z=1 (N=P=0) when the value stored in one of the registers is zero

## LC-3 instruction set




superscript "+" denotes instructions that update the condition bits NZP

# LC-3 assembly language

- LC-3 assembly language is a low-level language specifically invented for LC-3 computer
  - It is machine-specific; different processors have different assembly languages
  - Each assembly language instruction corresponds to a single ISA instruction
- Assembly language makes it much easier to program by using human-readable language instead of binary words, while still providing the programmer with the fine-grain control over the instructions and data
- Assembly language allows to refer to *instructions*, *memory locations*, and *values* **symbolically**.

## LC-3 Assembly language instruction

- Format
  - An instruction in LC-3's assembly language consists from 4 parts
    - (LABEL) OPCODE OPERANDS (COMMENT)
    - OPCODE and OPERANDS are required
    - LABEL and COMMENT are optional
- Opcodes
  - Are reserved symbols that correspond to LC-3 instructions (it is easier to remember a name than a sequence of 0s and 1s)
  - 15 opcodes in LC-3 ISA
    - ADD, AND, LD, LDR, etc.
- Operands
  - Each instruction has a number of instruction-specific operands, written in a particular order
  - Operands are one of the following:
    - Registers: Rn, where n is from 0 to 7
      - E.g.: ADD R2, R3, R4
    - Numbers: starting with # (decimal) or x (hex)
      - E.g.: AND R0, R0, #0
    - Labels: symbolic names of memory locations
      - BRnzp NEXT
  - Operands are separated by a comma
- Labels
  - Symbolic names used to identify memory locations that are explicitly referred to in the program
  - Consist of 1 to 20 alphanumeric characters
  - Used for two reasons:
    - To specify a memory location which is the target of a branch instruction
      - e.g., BRnzp TEST
    - To specify a memory location that contains a value loaded/stored by the LD/ST family of instructions
      - e.g., LD R3, PTR
      - here value stored in memory at the location labeled with PTR will be loaded into R3
- Comments
  - Intended only for human consumption, that is, ignored by the computer
  - Their purpose is to make the program more readable; comments should provide more insights, not just restate the obviously

- Text after semicolon (;) is considered to be a comment

## Pseudo-ops

- Do not refer to operations that will be actually executed by the computer
- They really are just assembler directives that are processed by the assembler during the assembler language translation into binary program representation
- LC-3 assembly language has 5 pseudo-ops
  - .ORIG specifies where in the memory the program should be placed
    - Format: `.ORIG <address>`
  - .END indicates the end of the program text. It has nothing to do with stopping the execution of the program, it just merely says that any text after it is not a part of the program and should be ignored
    - Format: `.END`
  - .FILL indicates that a location in memory will be filled with a value specified in the operand
    - Format: `.FILL <value>`
    - `E.g.:`

```
.ORIG x3000
.FILL xABCD
.FILL xFEDC
.END
```

    - This will result in value xABCD stored in memory at address x3000 and value xFEDE stored in memory at address x3001.
  - .BLKW (block of words) indicates that some number of memory locations, starting from the current location, is to be reserved
    - Format: `.BLKW <n>`
    - `E.g.:`

```
.ORIG x3000
.FILL xABCD
.BLKW #2
.FILL xFEDC
.END
```

    - This will result in value xABCD stored in memory at address x3000 and value xFEDE stored in memory at address x3003. Memory locations x3001 and x3002 will be reserved for a block of 2 words.
  - .STRINGZ indicates that the next few memory locations will be reserved and initialized with the characters provided in the operand
    - Format: `.STRINGZ "<text>"`
    - Example:

```
.ORIG x3000
.STRINGZ "test"
.END
```

    - This will result in the following values stored in memory starting from address x300:

```
x3000    't'
```

```
x3001    'e'
x3002    's'
x3003    't'
x3004    '\0'
```

- Note last value '\0', which is automatically added by the assembler; this is referred to as *null-terminated* string

## TRAP code pseudo-instructions

- "Standard" trap instructions are given more descriptive names
  - HALT, same as TRAP x25
  - IN, same as TRAP x23
  - OUT, same as TRAP 21
  - GETC, same as TRAP x20 (read one char from the keyboard)
  - PUTS, same as TRAP x22 (write null-terminated string to the keyboard)

## Assembly process

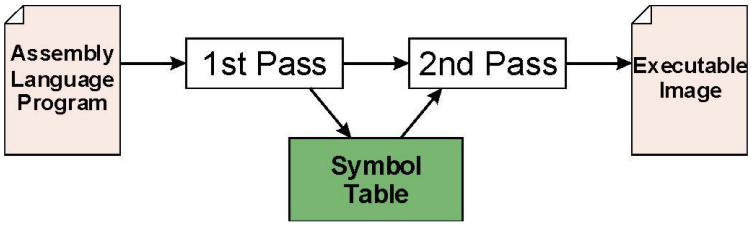
- We save the program into a file with extension .asm, e.g., **count.asm**
- Convert it to an LC-3 executable: **lc3asm count.asm :**

```
STARTING PASS 1
0 errors found in first pass
STARTING PASS 2
0 errors found in second pass
```

- lc3asm translated the program stored in count.asm into a binary code stored in **count.obj**, which we can now execute in the LC-3 simulator
- lc3asm program is called assembler, it translates a program written in the assembly language into a binary machine representation
- The assembly process is performed in 2 steps (passes)
  - Pass 1: scan the entire program and create the *symbol table* is created. Symbol table is a correspondence of symbolic names and their 16-bit memory addresses
    - In case of our count.asm program, symbol table looks as follows:

| Symbol | Address |
|---|---|
| NEXT | x3003 |
| MOVEON | x300B |
| OUTPUT | x300D |
| PTR | x300F |
| CHAR | x3010 |
| RESULT | x3011 |

  - Pass 2: machine language program is generated
    - Line-by-line, each assembly language instruction is converted into machine (binary) language instruction
    - Each time a symbol is found as one of instruction's operands, a value from the symbol table is used to resolve the memory address of the offset
- From assembler to a running program:

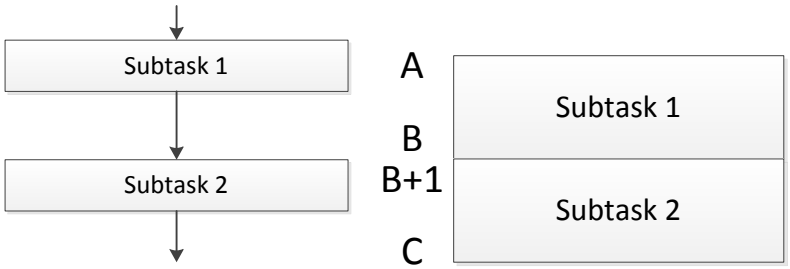



- Multiple object files
  - LC-3 simulator can load multiple object files into memory, then start executing at a desired address
    - system routines, such as keyboard input, are loaded automatically
    - loaded into "system memory," below x3000
  - user code should be loaded between x3000 and xFDFF
    - each object file includes a starting address
    - be careful not to load overlapping object files
- Linking and loading
  - Loading is the process of copying an executable image into memory
  - Linking is the process of resolving symbols between independent object files
    - suppose we define a symbol in one module, and want to use it in another
    - linker will search symbol tables of other modules to resolve symbols and complete code generation before loading

## Implementing programming constructs in LC-3 assembly language

### Sequential construct

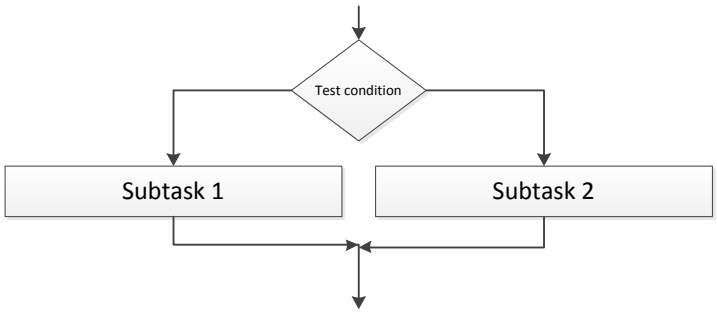
- Is used when we can decompose a given task into two smaller sub-tasks, such that one of them has to be fully executed before we can execute the other:
- Subtask 1 starts at memory address A and ends at memory address B
- Subtask 2 then starts at memory address B+1

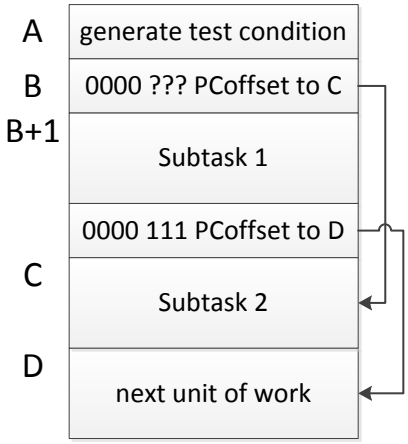



### Conditional construct

- Is used when the task consists of some subtasks such that only one of them needs to be executed, but not all, depending on some condition
  - If the test condition is true, we need to carry out one subtask, otherwise we carry out the other task; but we newer carry out both tasks
  - Once one of these subtasks is executed, we never go back
- In LC-3:
  - Create code that converts decision condition into one of the condition codes N,Z,P
  - Use conditional BR instruction to transfer control to the proper subtask

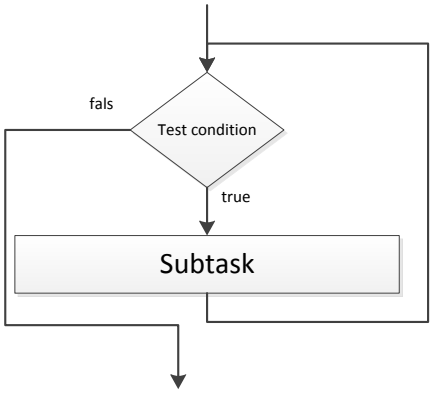
- Use unconditional BR to transfer control to the next task after first subtask is done

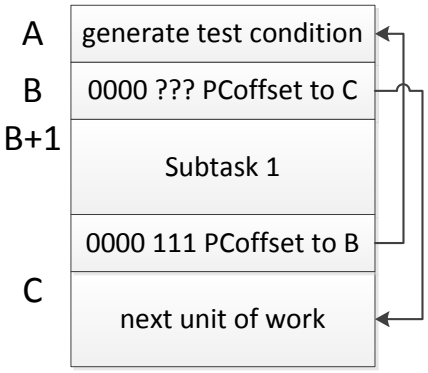



### Iterative construct

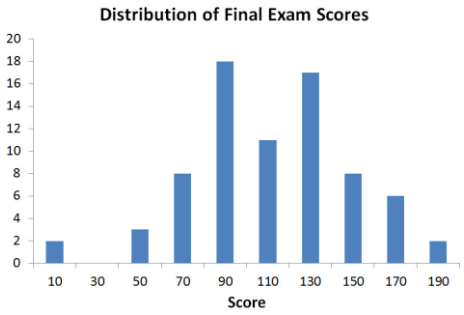
- Is used if one of subtasks needs to be re-done several times, but only as long as some condition is met
  - Each time we finish executing the subtask, we go back and reexamine the condition
  - The moment the condition is not met, the program proceeds onwards
- In LC-3:
  - Create code that converts decision condition into one of the condition codes N,Z,P
  - Use conditional BR instruction to branch if the condition generated is *false*
  - Use unconditional BR to transfer control back to the condition generating code after the subtask is done

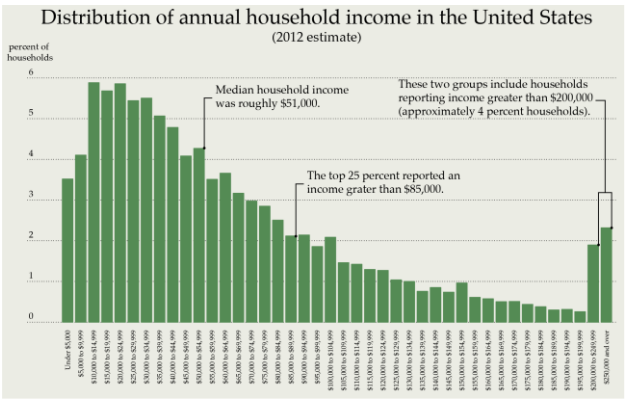



## Example: Computing a histogram

- Problem statement: Given a string or stream of data, e.g., sequence of numbers or ascii characters, compute the histogram or the frequency distribution of the different characters.
- What is a histogram?




- Specifics: Count the occurrences of each letter (A to Z) in an ASCII string terminated by a NUL character. Lower case and upper case should be counted together, and a count also kept of all non-alphabetic characters (not counting the terminal NUL).
  - String starts at x4000.
  - Histogram (NOT initialized) should be stored starting at x3F00, with the non-alphabetic count at x3F00, and the count for each letter in x3F01 (A) through x3F1A (Z).

| Address | Contents | Comments |
|---|---|---|
| | | |
| x3000 | .ORIG | ;program starts |
| … | | |
| | | |
| | | |
| x3F00 | | ; count for non alphabets |
| x3F01 | | ; count for ‘a’ and ‘A’ |
| x3F02 | | ; count for ‘b’ and ‘B’ |
| … | | |
| x3F1A | | ;count for z and Z |
| … | | |
| x4000 | ‘T’ | ; beginning of string |

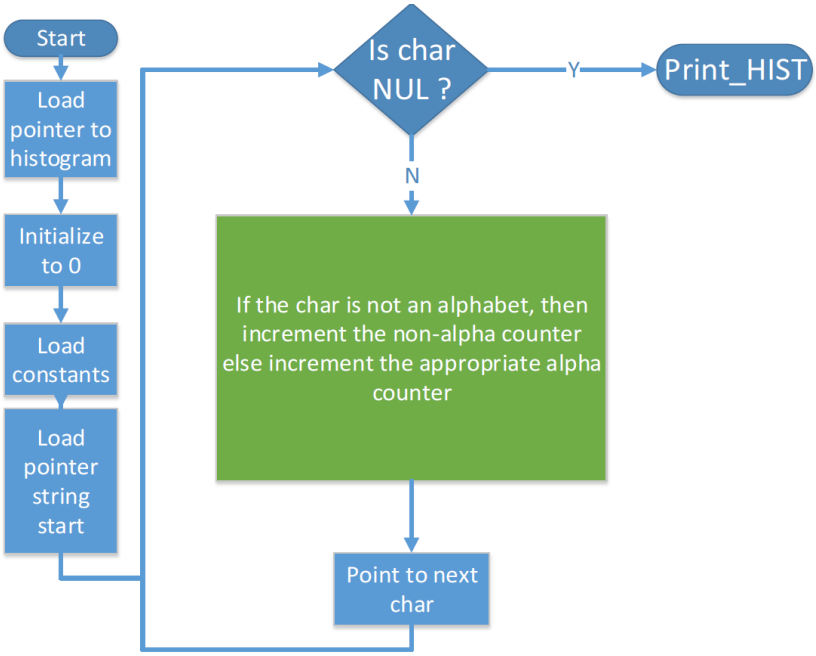
- Example input and output

Example Input and Output

| Address | Contents | Comments |
|---|---|---|
| | | |
| x4000 | ‘H’ | ;string starts |
| x4001 | ‘e’ | |
| | ‘l’ | |
| | ‘L’ | |
| | ‘o’ | |
| | ‘ ’ | |
| | ‘E’ | |
| | ‘c’ | |
| | ‘e’ | |
| | ‘2’ | |
| | ‘2’ | |
| | ‘0’ | |
| | ‘!’ | |
| | NUL | ;string ends |

| Address | Contents | Comments |
|---|---|---|
| | | |
| x3F00 | x0005 | ;count for non alpha |
| x3F01 | x0000 | ;count for a |
| | x0000 | ;count for b |
| | x0001 | ;count for c |
| | | |
| x3F05 | x0003 | ;e |
| | … | |
| | x0001 | ;h |
| | | |
| | x0002 | ;l |
| | | |
| | x0001 | ;o |
| | | |
| | | |

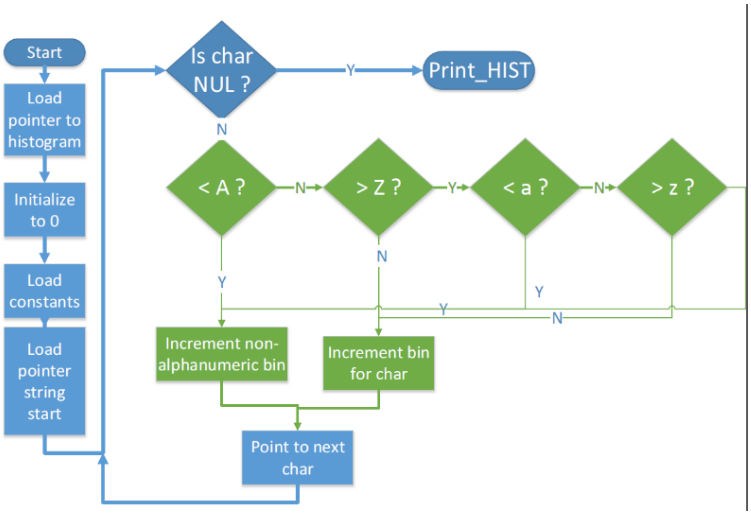
- Flowchart




- ASCII table

| DEC | HEX | CHAR | DEC | HEX | CHAR |
|---|---|---|---|---|---|
| 64 | 40 | @ | 96 | 60 | ` |
| 65 | 41 | A | 97 | 61 | a |
| 66 | 42 | B | 98 | 62 | b |
| 67 | 43 | C | 99 | 63 | c |
| 68 | 44 | D | 100 | 64 | d |
| 88 | 58 | X | 120 | 78 | x |
| 89 | 59 | Y | 121 | 79 | y |
| 90 | 5A | Z | 122 | 7A | z |

- Refined flowchart




- From the flowchart to code
  - Registers

| Register name | contents |
|---|---|
| R0 | pointer to histogram (x3100) |
| R1 | pointer to current position in string; loop count during initialization |
| R2 | current character; and points to the histogram entry |
| R3 | -'@' (0xFFC0) |
| R4 | '@'-'Z' = 'BQ' – 'z'(xFFE6) |
| R5 | '@' - 'BQ' (xFFE0) |
| R6 | temporary register |

- Overall program structure

| Address | Contents | Comments |
|---|---|---|
| | .ORIG x3000 | ; Program start |
| x3000 | ... | ; compute histogram |
| | | |
| | | |
| PRINT_HIST | … | ; print histogram |
| | | |
| | | |
| HALT | | Done |
| NUM_BINS | #27 | ; number of histogram bins |
| HIST_ADDR | x3F00 | ; start address of the histogram in memory |
| STR_START | x4000 | ; start address of string |
| | … | |
| | .END | |

- Part of the code to compute histogram

| Address | Contents | Comments |
|---|---|---|
| COUNTLOOP | LDR R2, R1,#0 | ;load next char x from string |
| | BRZ PRINT_HIST | ;end of string |
| | ADD R2, R2, R3 | ;R2 = x-'@' |
| | BRP AT_LEAST_A | if x > '@', i.e., x >= 'A' |
| NON_ALPHA | LDR R6, R0, #0 | ; load non-alpha count in R6 |
| | ADD R6, R6, #1; | ; increment non-alpha count |
| | STR R6, R0, #0; | ; store non-alpha count |
| | BRNZP GET_NEXT | ; jump to read next char |
| AT_LEAST_A | ADD R6, R2, R4 | ; compare x with Z<br>; R6 = x – '@' + ('@' – 'Z') = x – 'Z' |
| | BRp MORE_THAN_Z | ; branch if x > Z |
| ALPHA | ADD R2, R2, R0 | ; increment **appropriate** bin<br>; R2 = x – '@' + x3100 |
| | LDR R6, R2, #0 | ; load current count of x |
| | ADD R6, R6, #0 | ; increment count |
| | STR R6, R2, #0 | ; store count |
| | BRnzp GET_NEXT | |
| MORE_THAN_Z | | |